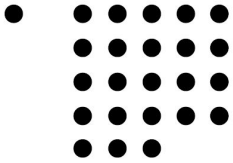


Fachhochschule Köln
Cologne University of Applied Sciences

Masterthesis

Real Time Tone Mapping

Eine Evaluierung farbgetreuer Verfahren zur Luminanzkompression

vorgelegt von

B. Eng. Sebastian Dille
Mat. Nummer: 11085105

im Studiengang

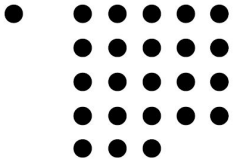
Media and Imaging Technology

betreut durch

Prof. Dr.-Ing. Arnulph Fuhrmann

Prof. Dr.-Ing. Gregor Fischer

27. Mai 2015



Fachhochschule Köln
Cologne University of Applied Sciences

Master Thesis

Real Time Tone Mapping

An evaluation of color-accurate methods for luminance compression

submitted by

B. Eng. Sebastian Dille
Mat. number: 11085105

in the graduate program

Media and Imaging Technology

reviewed by

Prof. Dr.-Ing. Arnulph Fuhrmann

Prof. Dr.-Ing. Gregor Fischer

May 27th, 2015

Inhaltsverzeichnis

1 Kurzfassung

Titel: Real Time Tone Mapping - Eine Evaluierung farbgetreuer Verfahren zur Luminanzkompression

Gutachter:
Prof. Dr.-Ing. Arnulph Fuhrmann
Prof. Dr.-Ing. Gregor Fischer

Zusammenfassung: Die neusten Fortschritte im Bereich Real Time Rendering ermöglichen virtuelle Produktionsabläufe in weiten Teilen der Industrie. Diese Vorgehensweise setzt latenzfreies Arbeiten und eine akkurate Erscheinung voraus. Daher empfiehlt sich die Verwendung von HDR-Rendering für fotorealistische Ergebnisse und Tone Mapping für die passende Darstellung.

Gleichwohl gibt es bisher nur wenige Publikationen, die sich mit echtzeitfähigem Tone Mapping beschäftigen. Zudem tendieren Tone Mapping Operatoren dazu, Farberscheinungseffekte zu erzeugen, die zu einer Diskrepanz zwischen dargestelltem und realem Produkt führen.

Deshalb werden in dieser Thesis sieben Tone Mapping Operatoren bewertet, die unterschiedliche Ansätze zur Farbkorrektur beinhalten. Sie werden als Fragment Shader implementiert, um Echtzeitverarbeitung zu realisieren. Zusätzlich wird ihre Performance und die subjektive Akkuratesse der Ergebnisse gemessen.

Im Ergebnis schneidet der FilmicTMO als global arbeitender Sigmoid-Operator am besten ab und wird für die weitere Verwendung empfohlen.

Stichwörter: Tone Mapping; Real Time Rendering;

Datum: 27. Mai 2015

Abstract

Title: Real Time Tone Mapping - An evaluation of color-accurate methods for luminance compression

Reviewers:
Prof. Dr.-Ing. Arnulph Fuhrmann
Prof. Dr.-Ing. Gregor Fischer

Abstract: Recent advances in real time rendering enable virtual production pipelines in a broad range of industries. These pipelines require a fast, latency-free handling in addition to the accurate appearance of results. This suggests the use of HDR rendering for photorealistic results and a tone mapping operator for a matched display.

However, tone mapping of real time rendering has not been the focus of many publications to this day. Further more, tone mapping tends to cause image appearance effects, which leads to a change in the perception of colors and therefore a discrepancy between displayed and real world products.

For this purpose, seven tone mapping operators with different approaches of color correction are evaluated in this thesis. They are implemented as fragment shader to realise real time processing. In addition, their performance and subjective accuracy is measured.

As a result, the FilmicTMO as global sigmoid-based operator proceeds best and can be recommended for future work.

Keywords: Tone Mapping; Real Time Rendering;

Date: May 27th, 2015

2 Zielsetzung

In weiten Bereichen der Industrie hält seit einigen Jahren das Prinzip der virtuellen Produktion Einzug. Die Entwicklung von neuen Produkten wird zunächst an digitalen Modellen betrieben, die einfach und kostengünstig zu verändern sind. So können die ersten Entwicklungs- und Designschritte virtuell umgesetzt werden, bevor für deutlich höhere Kosten ein realer Prototyp gefertigt wird (vgl. z.B.[72]).

Die Grundvoraussetzung für dieses Prinzip ist die Verlässlichkeit des virtuellen Modells. Dieses muss sowohl in seinem Verhalten als auch in der Erscheinung exakt mit der Referenz übereinstimmen, damit die virtuellen Bearbeitungsschritte auch in der Realität übernommen werden können. Die korrekte Simulation des Verhaltens ist nicht Bestandteil dieser Arbeit. Der Fokus liegt stattdessen auf der korrekten Erscheinung.

Eine realistische Erscheinung setzt physikalisch korrektes Rendering voraus. Dieses muss zudem in Echtzeit geschehen, um eine flüssige Bedienung zu ermöglichen. Durch physikalisch korrektes Rendering werden jedoch zwangsläufig Helligkeitswerte erzeugt, die denen der Natur entsprechen und die ohne Tone Mapping nicht darstellbar sind (vgl. Kapitel 4 auf Seite 23 und Kapitel 5 auf Seite 29). Es muss also zusätzlich ein Tone Mapping Verfahren zum Einsatz kommen, durch welches der gesamte Prozess die Echtzeitanforderungen immer noch erfüllt.

Durch das Tone Mapping selbst entsteht jedoch eine weitere Problematik: Das Herabsetzen der Helligkeit erzeugt die in Abschnitt 3.3.2 auf Seite 14 beschriebenen Effekte, welche wiederum die Farberscheinung verändern. Das entspräche einer Abweichung von der Referenz, die vermieden werden soll.

Das Ziel dieser Arbeit ist daher die Entwicklung eines echtzeitfähigen, farbgetreuen Tone Mapping Verfahrens und seine Evaluierung.

3 Grundlagen

Die Wahrnehmung von Bildern im Allgemeinen und von Farbe im Speziellen sind komplexe Vorgänge, die von unterschiedlichen Faktoren beeinflusst werden. Zum besseren Verständnis wird hier zunächst auf die Grundlagen des menschlichen Sehens einerseits und der Erzeugung von Bildern andererseits eingegangen.

3.1 Licht

Die Grundlage des menschlichen Sehens bildet Licht. Es handelt sich um elektromagnetische Strahlung, die von einer Lichtquelle ausgesendet wird. Erreicht diese Strahlung unser Auge, können wir durch verschiedene Mechanismen (siehe Abschnitt 3.2 auf Seite 7) das Licht in visuelle Informationen umsetzen. So ist es möglich, Objekte in unserer Umgebung zu erkennen, die das Licht auf seinem Ausbreitungsweg getroffen hat und die wir anhand ihrer Reflexions- bzw. Transmissionseigenschaften voneinander unterscheiden können.

3.1.1 Radiometrische Größen

Da Licht eine Strahlungsart ist, gelten die radiometrischen Größen in Tabelle 3.1. Sie finden vor allem bei generellen Beschreibungen Verwendung, welche unabhängig vom jeweiligen Betrachter sind.

Größe	Bedeutung	Einheit
Strahlungsfluss	Strahlungsenergie pro Zeit	W
Strahlungsenergie	Energie einer Anzahl Photonen	J
Strahlungsstärke	Strahlungsfluss pro Raumwinkel	$\frac{W}{sr}$
Bestrahlungsstärke	Strahlungsfluss pro Fläche	$\frac{W}{m^2}$
spezifische Ausstrahlung	Strahlungsfluss pro Senderfläche	$\frac{W}{m^2}$
Bestrahlung	Strahlungsenergie pro Fläche	$\frac{J}{m^2}$
Strahldichte	Strahlungsfluss pro Raumwinkel und Senderfläche	$\frac{W}{m^2 sr}$

Tabelle 3.1: Radiometrische Einheiten

3.1.2 Fotometrische Größen

Die meisten Lichtquellen emittieren jedoch nicht nur eine einzige Wellenlänge, sondern ein Spektrum verschiedener Frequenzen. In solchen Fällen ist die Verwendung radiometrischer Größen unzureichend, da sie den Bezug zur jeweiligen Wellenlänge außer Acht lassen. Gerade bei Anwendungen, welche die Empfindungen des Betrachters mit berücksichtigen, spielt die Wellenlänge jedoch eine entscheidende Rolle, da Menschen für unterschiedliche Wellenlängen unterschiedlich empfindlich sind.

Normbetrachterkurven Im Jahr 1924 hat die Beleuchtungskomission CIE, die Commission Internationale de l'Éclairage, 7 Studien durchgeführt, um die Empfindlichkeit des menschlichen Auges auf die verschiedenen Wellenlängen zu untersuchen. Die Ergebnisse wurden über die Anzahl der Teilnehmer gemittelt, um Werte für einen sogenannten Normbetrachter zu erhalten. Dieser Normbetrachter ist jedoch rein virtuell, sodass es theoretisch sein kann, dass kein Mensch die exakt gleichen Empfindlichkeitskurven aufweist [53] [63].

Multipliziert man die radiometrischen Größen mit den Normbetrachterkurven, erhält man die fotometrischen Größen in Tabelle 3.2, welche damit abhängig von der Wellenlänge bzw. dem Spektrum der Lichtquelle sind.

Größe	Bedeutung	Einheit
Lichtstrom	Strahlungsleistung einer Lichtquelle	lm
Lichtmenge	Strahlungsenergie einer Lichtquelle	$lm * s$
Lichtstärke	Lichtstrom pro Raumwinkel	cd
Beleuchtungsstärke	Lichtstrom pro beleuchteter Fläche	lx
spezifische Lichtausstrahlung	emittierter Lichtstrom	lx
Belichtung	Beleuchtungsstärke pro Zeit	$lx * s$
Leuchtdichte	Lichtstärke bezogen auf die Größe der Quelle	$\frac{cd}{m^2}$

Tabelle 3.2: Fotometrische Einheiten

3.1.3 Lichtarten

In Abhängigkeit vom oben erwähnten Spektrum können verschiedene Lichtarten definiert werden. Die gebräuchlichsten sind die in ISO 3664 angegebenen Normlichtarten [31].

Darunter befindet sich die Lichtart A, die dem Frequenzspektrum einer Kerze mit einer Farbtemperatur von 2865K entspricht, und die F-Reihe, in der Spektren verschiedener Leuchtstoffröhren festgelegt sind. Für Tageslicht existiert die D-Reihe, deren Spektren auch ultraviolettes Licht enthalten.

D65: Bei der D65-Lichtart handelt es sich um einen Vertreter der D-Reihe, der in den meisten in Kapitel 5 auf Seite 29 und Kapitel 7 auf Seite 39 vorgestellten Verfahren eingesetzt wird. D65 entspricht dem Tageslichtspektrum bei einer Farbtemperatur von 6500 Kelvin. Dies ist beispielsweise bei bewölktem Mittagshimmel gegeben.

3.1.4 Dynamikumfang

Das Verhältnis von hellster zur dunkelster Stelle einer Szene wird als Dynamikumfang bezeichnet. Abhängig von den herrschenden Lichtverhältnissen kann er stark variieren.

Tabelle 3.3 zeigt typische Leuchtdichten für verschiedene Lichtquellen [34].

Lichtquelle	mittlere Leuchtdichte ($\frac{cd}{m^2}$)
Mittagssonne	bis $1,6 * 10^9$
Fotoblitz	160000000 bis 400000000
Abendsonne	6000000
Glühlampe, mattiert	50000 bis 500000
Kerze	7000
Mond	2500

Tabelle 3.3: Typische Leuchtdichten

Betrachtet man die entsprechenden Beleuchtungsstärken in Tabelle 3.4 [27], ist erkennbar, wie sehr sich dieses Verhältnis abhängig von der Umgebung verändert.

Situation	Beleuchtungsstärke (lx)
Sonnenschein	bis 100000
Bewölkter Himmel	1000 bis 10000
Wohnung	100 bis 200
Straßenbeleuchtung	1 bis 20
Kerzenschein	1
Vollmond	0,25
Mondlose Nacht	0,01

Tabelle 3.4: Typische Beleuchtungsstärken

Besonders große Werte entstehen, wenn unterschiedliche Umgebungssituationen in einer Szene vereint sind. Dies ist z.B. der Fall, wenn in einem schwach erleuchteten Zimmer der Himmel durch ein Fenster zu sehen ist.

Der Dynamikumfang spielt dann eine Rolle, wenn ein Bild erzeugt, gespeichert oder wiedergegeben werden soll, denn alle heute existierenden bildgebenden Verfahren können grundsätzlich nur einen begrenzten Dynamikumfang darstellen.

So verfügt ein analoges Filmnegativ beispielsweise über eine endliche Anzahl von Silbermolekülen, sodass ab einer bestimmten Lichtmenge die Filmemulsion bereits gesättigt ist und keine weitere Reaktion durch eine noch größere Lichtmenge mehr möglich ist [10].

In der Digitaltechnik ergibt sich die Begrenzung hauptsächlich durch fehlenden Speicherplatz bzw. eine für die notwendige Bildverarbeitung zu geringe Bandbreite. Beides ist notwendig, um sowohl sehr große Werte zu speichern, als auch diese Werte in sehr kleine Abstufungen einteilen zu können, um feine Unterschiede in den hellen und dunklen Bereichen zu erhalten [66].

In der Praxis bedeutet dies z.B., dass ein Fotograf sich über die Wahl von Belichtungszeit und Blende zwischen der korrekten Abbildung der dunklen oder der hellen Stellen des Motivs entscheiden muss. Diesbezüglich wird der Dynamikumfang meist in Belichtungswerten (engl.: „exposure values“, kurz „EV“) angegeben.

3.2 Das visuelle System

Das menschliche Auge funktioniert ähnlich wie eine Fotokamera. Licht fällt durch die Linse auf die Netzhaut. Die dort sitzenden Photorezeptoren nehmen das Licht wie die Pixel eines Sensors auf und wandeln die Strahlung in elektrische Impulse um. Diese gelangen über den Sehnerv ins Gehirn und werden dort verarbeitet.

Ist die Lichtmenge zu groß, wird die Öffnung des Auges durch die Iris verengt. Analog zu der Blende einer Kamera wird so die einfallende Lichtmenge reduziert. Der Einfluss der Iris reicht jedoch nicht aus, um die vollständige Adaption des Auges an verschiedene Lichtverhältnisse zu erklären, die für den großen im Auge verarbeitbaren Dynamikumfang verantwortlich ist [53].

3.2.1 Stäbchen und Zapfen

Auf der Netzhaut des Auges befinden sich zwei Arten von Photorezeptoren, die aufgrund ihrer Form als Stäbchen und Zapfen bezeichnet werden [53]. Den überwiegenden Teil nehmen die Stäbchen mit ca. 120 Millionen Zellen ein. Sie sind nur für einen bestimmten Frequenzbereich empfindlich und sorgen damit ausschließlich für monochromatisches Sehen. Die Empfindlichkeit der Stäbchen in diesem Frequenzband ist jedoch sehr hoch, sodass sie auch bei sehr wenig Licht noch Informationen liefern können.

Die Zapfen stellen mit ca. 7 Millionen den kleineren Anteil der Photorezeptoren. Sie sind weniger empfindlich als die Stäbchen, reagieren aber auf einen größeren Teil des elektromagnetischen Spektrums. Zapfen lassen sich wiederum in drei Arten unterteilen, die jeweils für einen anderen Frequenzbereich besonders empfindlich sind. Diese Einteilung in L-(Long), M-(Medium) und S-(Short)Zapfen ist der Grund für die Fähigkeit des Menschen, Farben wahrzunehmen. Diese Zapfenarten senden insgesamt drei unterschiedliche Signale ans Gehirn, weswegen oft der Begriff Tristimulus verwendet wird.

Interessanterweise lässt sich zeigen, dass der Mensch zwar ein rötliches Blau oder ein bläuliches Grün wahrnehmen kann, aber ein rötliches Grün in der Wahrnehmung nicht existiert. Daher bestand lange Zeit eine zweite Theorie, derzufolge sich die Zapfen nicht durch trichromatische Signale sondern durch sogenannte Color-Opponent-Signale voneinander unterscheiden. Demnach habe ein Zapfen eine positive Anregung für rotes Licht und eine Art negative oder dämpfende Anregung für grünes Licht.

Untersuchungen mittels im Auge angebrachter Elektroden haben ergeben, dass beide Theorien zum Teil zutreffen. Die Zapfen reagieren analog zur Tristimulus-Theorie mit einer jeweils anderen Empfindlichkeitskurve. Die LMS-Signale werden in den darunter liegenden Zellen jedoch zunächst in ein Color-Opponent-Signal umgewandelt, bevor sie über den Sehnerv an das Gehirn weiter geleitet werden.

3.2.2 Adaption

Es existieren verschiedene Mechanismen, über die sich Stäbchen und Zapfen auf unterschiedliche Lichtsituationen einstellen können. Diese werden als Adaption bezeichnet. Man unterscheidet zwischen Hell- bzw. Dunkeladaption zur Anpassung an verschiedene Helligkeiten und chromatischer Adaption zur Anpassung an unterschiedliche Lichtspektren.

Hell-/Dunkeladaption

Als Hell- bzw. Dunkeladaption werden zwei Vorgänge bezeichnet, mit denen sich das Auge auf größere Wechsel in der Umgebungshelligkeit einstellt [53]. Sie ergänzen somit den Einfluss der Iris. Helladaption findet statt, wenn der Mensch von einer dunklen in eine helle Umgebung kommt, z.B. aus einem Tunnel raus ans Tageslicht tritt. Zum einen werden jetzt die Zapfen aktiv, während die Stäbchen inaktiv werden. Zum anderen senkt sich die Konzentration des Sehfarbstoffs, sodass das Auge unempfindlicher für Licht wird.

Der umgekehrte Mechanismus wird als Dunkeladaption bezeichnet. Er dauert entschieden länger, da neuer Sehfarbstoff produziert werden muss. So können mehrere Minuten vergehen, bis das Auge vollständig an die dunkle Umgebung angepasst ist.

Chromatische Adaption

Die Anpassung an verschiedene Lichtspektren wird als chromatische Adaption bezeichnet. Durch diesen Vorgang erscheint die Farbe einer Oberfläche bei unterschiedlichen Beleuchtungen konstant. So kann ein Blatt Papier sowohl unter Kunstlicht als auch bei Tageslicht als weiß wahrgenommen werden, obwohl im letzteren Fall der Anteil an kleinen Wellenlängen im Licht deutlich höher ist und somit ein Blaustich zu beobachten sein sollte.

Der Vorgang der chromatischen Adaption kann auch in der Fotografie umgesetzt werden. Dies wird als Weißabgleich bezeichnet [53].

3.3 Farbe

Wird Licht an einer Oberfläche reflektiert und erreicht unser Auge, entsteht durch die Reaktionsweise der Zapfen eine Farbe. Dabei handelt es sich um eine Art Empfindung, die uns hilft, Objekte anhand der reflektierten Wellenlängen voneinander zu unterscheiden. Sie wird durch viele verschiedene Faktoren beeinflusst, wie z.B. die Art der Oberfläche,

die daraus resultierenden Wellenlängen des reflektierten Lichts, das Grundspektrum der Lichtquelle, die Lichtintensität, die Hintergrundhelligkeit usw..

Da es sich um Empfindungen handelt, sind Farben selbst physikalisch nicht messbar. Es können zwar fotometrische und radiometrische Größen des Lichts bestimmt werden, wie die Lichtstärke oder die Wellenlänge, die Verarbeitung einer Kombination dieser Größen zu einer Farbe findet jedoch im Gehirn statt und kann durch Modelle nur angenähert werden [30].

Zudem sind Farben nicht immer eindeutig bestimmten Frequenzen zuzuordnen. Fast alle Oberflächen reflektieren nicht nur eine einzige, sondern ein Spektrum aus mehreren Wellenlängen. Es gibt also sehr viele verschiedene Kombinationen unterschiedlicher Wellenlängen, wovon einige im Gehirn eine identische Farbempfindung erzeugen. Dieser Sachverhalt wird Metamerie genannt (siehe ebenda).

3.3.1 Farbräume

Farbräume (engl. Color Spaces) stellen ein System dar, um die Empfindung „Farbe“ anhand bestimmter Merkmale zu ordnen [51]. Diese Merkmale spannen dabei einen Raum auf, dessen Dimensionen durch die Anzahl der Merkmale definiert sind. Dabei steht jeder Punkt in diesem Raum für eine Farbe. Nimmt man das System der additiven Farbmischung als Grundlage, können Farben zum Beispiel anhand ihrer Anteile an den drei Primärfarben Rot, Grün und Blau definiert bzw. voneinander unterschieden werden. Der so definierte Farbraum heißt RGB-Farbraum.

Die meisten Dateiformate speichern für jedes Pixel mehrere Werte, die sogenannten Farbkanäle, die i.d.R. den jeweiligen Primärfarben entsprechen.

Gamma: In einigen Farbräumen verhalten sich die Farbwerte nicht linear zur Helligkeit. Dadurch soll die Helligkeitswahrnehmung des Auges nachempfunden werden, die sich ebenfalls nicht linear verhält [54].

Bei der Umrechnung von linearen in nichtlineare Werte wird eine sogenannte Gamma-Korrektur angewandt. Diese entspricht einer Exponentialfunktion der Form

$$I_{nichtlinear} = I_{linear}^{\frac{1}{\gamma}}. \tag{3.1}$$

Hier ist Gamma der Kehrwert des Exponenten, der die Krümmung der Übertragungskurve angibt.

Von den vielen existierenden Farbräumen sollen einige hier besonders erwähnt werden, da sie in dieser Thesis und der dazugehörigen Software Verwendung finden:

XYZ-Farbraum

Der CIE-1931-XYZ-Farbraum [36] ist neben dem CIE-RGB-Farbraum der erste und heutzutage wichtigste Tristimulus-Farbraum, auch Normvalenzsystem genannt. Als solcher sind seine genormten Empfindlichkeitskurven ähnlich, jedoch nicht identisch

zu den Reaktionskurven der menschlichen Zapfen. Er ist speziell auf das menschliche Sehen abgestimmt und umfasst alle menschlich wahrnehmbaren Farben.

Im Unterschied zu dem als Basis dienenden CIE-RGB-Farbraum sind die Empfindlichkeitskurven so genormt, dass sie keine negativen Werte aufweisen. Dies ist jedoch nur mit imaginären Primärfarben realisierbar, welche selbst nicht erzeugbar bzw. sichtbar sind. Somit handelt es sich beim XYZ-Farbraum um ein rein mathematisches System.

Als zusätzliches Merkmal sind die Y-Werte so genormt, dass ihr Verlauf annähernd exakt dem Helligkeitsempfinden des Menschen entspricht. Dadurch werden auf der XZ-Ebene zu jedem Y-Wert alle Farbarten abgebildet, die unter der gegebenen Helligkeit entstehen können.

CIE-Normfarbtafel: Der dreidimensionale XYZ-Farbraum lässt sich schlecht in einem zweidimensionalen Bild darstellen. Aus diesem Grund wurde die Normfarbtafel entwickelt, auch Chromatizitätsdiagramm oder xy-Diagramm genannt [53]. Dieses Diagramm bildet quasi die XY-Ebene des XYZ-Farbraums ab. Die dritte Koordinate kann aus den übrigen beiden Werten durch die Beziehung

$$x + y + z = 1 \tag{3.2}$$

ermittelt werden.

Abbildung 3.1 auf der nächsten Seite zeigt das xy-Diagramm. Ebenfalls eingetragen ist die Black-Body-Kurve, auf der die verschiedenen Weißpunkte abgebildet sind. Durch die unterschiedlichen Spektren kommt es beim Wechsel der Lichtart in einer Szene zu einer Veränderung der wahrgenommenen Farben. Zu jeder Lichtart ist daher ein Weißpunkt festgelegt. Er gibt denjenigen Punkt im xy-Diagramm an, der unter der gegebenen Lichtart weiß erscheint. Dieser Weißpunkt kann bei der Transformation zwischen verschiedenen Farbräumen oder beim Gamut Mapping verwendet werden, um den Farbeindruck konstant zu halten oder an eine bestimmte Lichtart anzupassen.

Es gibt auch einige Farbräume, die einen fest definierten Weißpunkt besitzen, wie z.B. der sRGB-Farbraum.

Yxy-Farbraum

CIE Yxy ist ein Farbraum, welcher auf der Trennung zwischen Helligkeit (Luminanz) und Buntheit (Chrominanz) beruht [35]. Er besteht aus einer Achse für die Luminanz-Werte (Y) und zwei Buntheitsachsen (x und y). Letztere sind jedoch weder einer konkreten Farbe zuzuordnen noch handelt es sich bei CIE Yxy um ein Color-Opponenten-System.

Auf CIE Yxy aufbauend existieren weitere Luminanz-Chrominanz-Farbräume, wie CIE Luv oder CIE L*a*b*. Diese sind mit der Intention entstanden, die Abstände im Chromazitätsdiagramm an die tatsächlich wahrnehmbaren Farbabstände anzugleichen, was bei CIE Yxy nicht der Fall ist.

HPE/LMS-Farbraum

Der LMS-Farbraum gehört ebenfalls zu den Modellen, welche direkt auf das menschliche Sehen abgestimmt sind. Seine Empfindlichkeitskurven sind zu diesem Zweck genauer

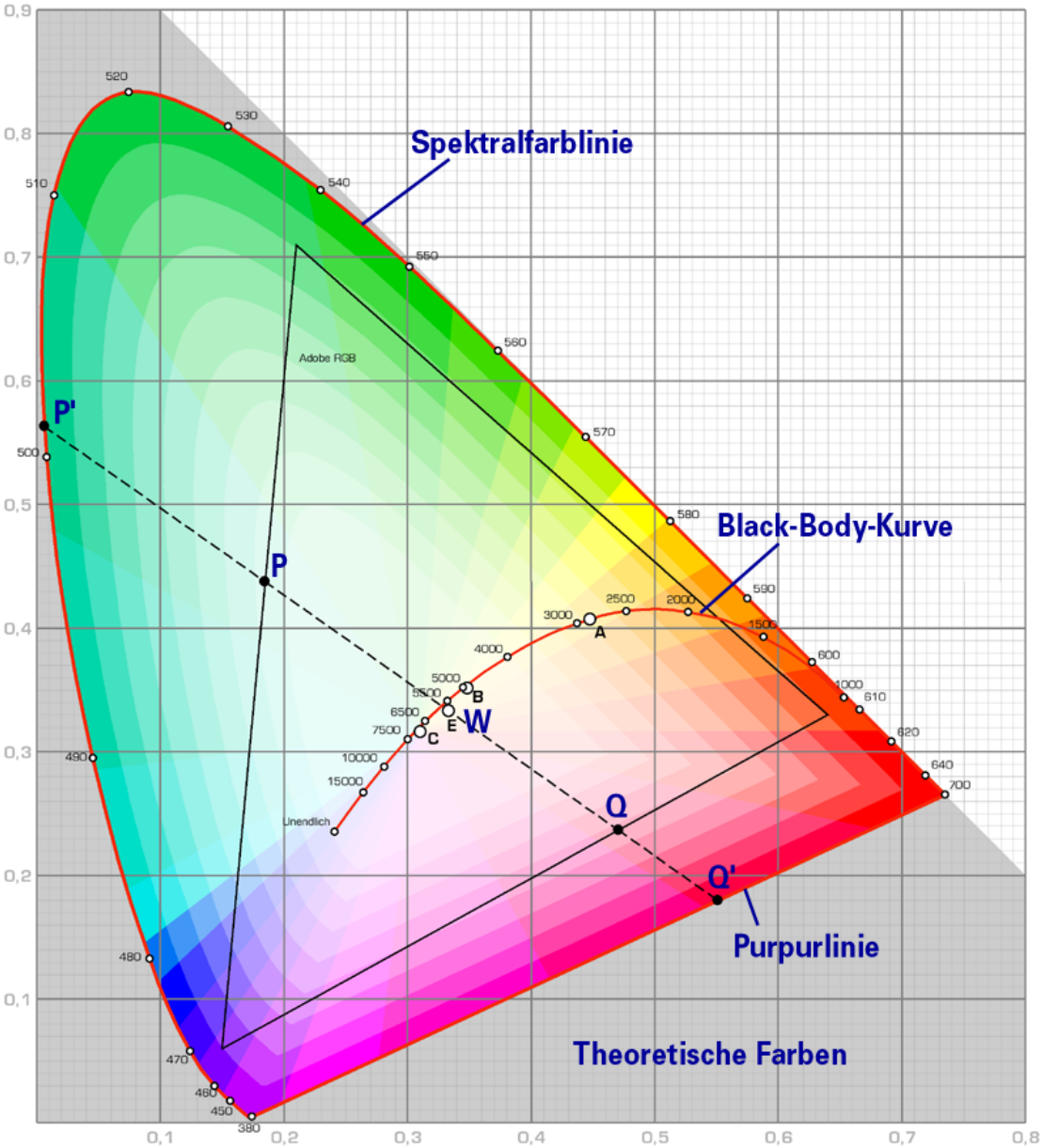


Abbildung 3.1: CIE-Normfarbtafel, Bildquelle: [4]

auf die Reaktionskurven der menschlichen Photorezeptoren angepasst als die des CIE-XYZ-Farbraums, weswegen er oft als Ausgangspunkt für Algorithmen und Modelle verwendet wird, welche das menschliche Sehen simulieren sollen, wie z.B. CIECAM02 [20].

Ein Farbwert, welcher in XYZ-Koordinaten vorliegt, kann durch eine sogenannte Hunt-Pointer-Estevez-Transformation in den LMS-Farbraum transformiert werden [66]:

$$\begin{pmatrix} L \\ M \\ S \end{pmatrix} = \begin{pmatrix} 0.3897 & 0.6890 & -0.0787 \\ -0.2298 & 1.1834 & 0.0464 \\ 0.0000 & 0.0000 & 1.0000 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}. \tag{3.3}$$

IPT-Farbraum

Bei dem IPT-Farbraum handelt es sich um einen Farb-Opponenten-Farbraum [19]. Diese Modelle greifen die Farbopponenten-Theorie auf und trennen analog zu Yxy die Information in Luminanz und Chrominanz, wobei die Farbwerte auf (i.d.R. zwei) Farbachsen arrangiert werden: eine Rot-Grün-Achse und eine Blau-Gelb-Achse. Eine Farbe kann also, in diesem Farbraum repräsentiert, nicht gleichzeitig „sehr grün“ und „sehr rot“ sein.

Obwohl IPT eigentlich für „image processing transform“ steht, könnten die Buchstaben laut Reinhard et al. [63] auch als Abkürzung für die drei Achsen betrachtet werden: intensity für die Luminanz, protan für Rot-Grün und tritan für Blau-Gelb.[1]

Um einen Farbwert von XYZ-Koordinaten in den IPT-Raum zu transformieren, wird zunächst die entsprechende LMS-Koordinate berechnet. Die einzelnen Werte für L, M und S werden dann über einen Exponenten von 0,43 nichtlinear komprimiert, bevor sie über eine weitere Matrix-Multiplikation in den IPT-Farbraum umgerechnet werden:

$$\begin{pmatrix} I \\ P \\ T \end{pmatrix} = \begin{pmatrix} 0.4000 & 0.4000 & 0.2000 \\ 4.4550 & -4.8510 & 0.3960 \\ 0.8056 & 0.3572 & -1.1628 \end{pmatrix} \begin{pmatrix} L' \\ M' \\ S' \end{pmatrix}. \tag{3.4}$$

Dabei gilt

$$L' = \begin{cases} L^{0.43}, & \text{falls } L \geq 0 \\ -(-L)^{0.43} & \text{sonst} \end{cases} \tag{3.5}$$

analog für M und S.

sRGB-Farbraum

sRGB steht für Standard-RGB. Es handelt sich hierbei um einen Farbraum, der 1996 in einer Kooperation von HP und Microsoft entwickelt wurde und speziell für die Verwendung im Internet gedacht ist [78]. Anders als beispielsweise CIE-XYZ hat er jedoch nur einen geringen Bezug zum visuellen System des Menschen. Er ist vielmehr darauf ausgelegt, von allen der damals üblichen CRT-Monitoren optimal dargestellt werden zu können, ohne komplexe ICC-Profile anwenden zu müssen.

[1] Die Begriffe protan und tritan entstammen der Medizin. Sie dienen als Kennzeichnung verschiedener Typen von Farbblindheit, wobei sich protan (von griechisch ”protos”: erster) auf einen Defekt des ersten Zapfentyps und tritan (von griechisch „tritos“: dritter) auf den dritten Zapfentyp bezieht [84].

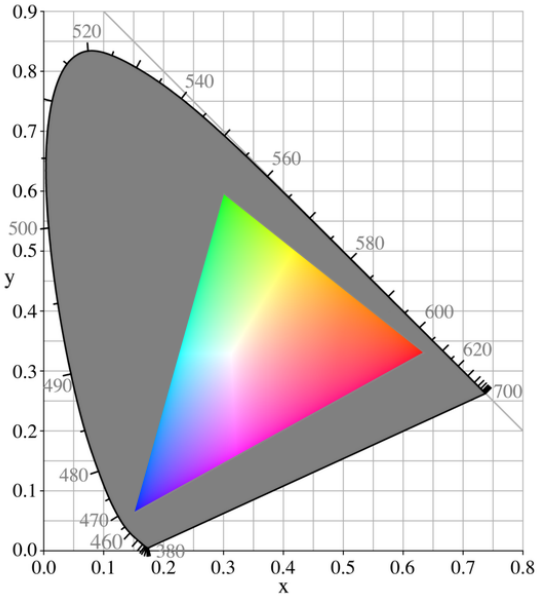


Abbildung 3.2: sRGB-Farbraum, Bildquelle: [75]

Aus diesem Grund werden die in ITU-R BT.709-3 definierten XYZ-Koordinaten als Primärfarben verwendet, dem vorher definierten HD-Fernseh-Standard (siehe Tabelle 3.5):

	x	y	z
R	0,64	0,33	0,03
G	0,30	0,60	0,10
B	0,15	0,06	0,79

Tabelle 3.5: sRGB-Primärfarben

Als Weißpunkt wird der D65-Weißpunkt gewählt, als Gamma ist ein Wert von ca. 2,2 definiert.

Wie aus den Primärfarben zu erkennen ist, sieht sRGB die Verwendung von normalisierten Werten vor. Werte kleiner als 0 oder größer als 1 werden in der Regel, je nach Implementierung, geclipped.

Abbildung 3.2 zeigt den Bereich der CIE-Normfarbtafel, der durch den sRGB-Farbraum erfasst wird. Es ist gut zu erkennen, dass letzterer deutlich kleiner ist als der XYZ-Farbraum.

HSL-Farbraum

Der HSL-Farbraum gehört zu einer Gruppe von Farbräumen, mit welchen versucht wird, die Erscheinung einer Farbe darzustellen [51]. Daher werden hier nicht drei Grundfarben oder Empfindlichkeitskurven als Basis verwendet, sondern der Farbton (engl. hue), die Sättigung (engl. saturation) sowie die relative Helligkeit (engl. lightness). Vergleichbar dazu sind der HSI-Farbraum mit der Intensität I (engl. intensity) und der HSV-Farbraum mit dem Wert V (engl. value).

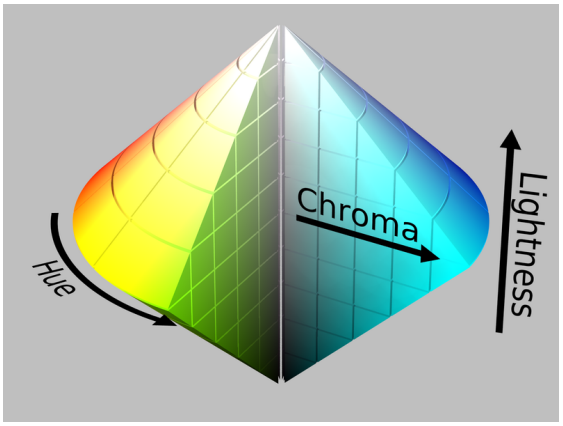


Abbildung 3.3: HSL-Farbraum, Bildquelle: [70]

Alle drei Farbräume haben jedoch gemeinsam, dass die verwendeten Attribute nicht genau mit den tatsächlichen Farbwahrnehmungsattributen (siehe Abschnitt 3.3.2) übereinstimmen.

3.3.2 Farberscheinung

Die menschliche Empfindung der Farbe kann nicht alleine durch Betrachtung des von einer Oberfläche reflektierten Lichts beschrieben werden. Vielmehr spielen eine Reihe anderer Faktoren, wie das Umgebungslicht und die Lichtstärke, eine entscheidende Rolle. Ändert sich einer dieser Faktoren, erscheint auch die Farbe einer Oberfläche anders, obwohl sich an der Oberfläche selbst nichts verändert hat. In diesem Zusammenhang spricht man von Farberscheinung (s. Hunt [30]).

Farberscheinungsattribute

Nach Hunt existieren verschiedene Attribute, um die tatsächliche Erscheinung einer Farbe zu charakterisieren:

Brightness beschreibt die absolute Helligkeit einer Fläche, also in welchem Maße eine Fläche Licht zu emittieren scheint.

Lightness beschreibt ebenfalls die Helligkeit einer Fläche, allerdings in Relation zu einer anderen Fläche, welche unter der gleichen Beleuchtung weiß erscheint. Lightness kann daher als relative Helligkeit angesehen werden [21].

Hue bzw. Farbton beschreibt den visuellen Eindruck, eine Fläche sei in ihrer Erscheinung ähnlich dem Eindruck einer der Farben Rot, Grün, Blau oder Gelb.

Colorfulness oder Farbigkeit gibt den Eindruck wieder, inwiefern eine Fläche sich visuell von Grau ($\hat{=}$ Fehlen von Farbe) unterscheidet.

Chroma oder Buntheit entspricht dem Eindruck von Farbigkeit in Bezug zu einer zweiten Fläche, welche unter der gleichen Beleuchtung weiß erscheint. Ähnlich wie

Lightness stellt Chroma ein relatives Attribut dar.

Saturation oder Sättigung entspricht dem Eindruck von Farbigkeit einer Fläche in Bezug zu ihrer Helligkeit. Sie gibt an, inwiefern sich eine Farbe bei gleichbleibender Helligkeit von Grau unterscheidet.

Mit Hilfe dieser Attribute kann die Erscheinung einer Farbe eindeutig beschrieben werden. Sie können daher genutzt werden, um bei der Transformation von einem Farbraum in den anderen die Farberscheinung zu erhalten. Zu diesem Zweck existieren verschiedene sogennante Gamut Mapping Algorithmen, welche in Kapitel 6 auf Seite 36 beschrieben werden.

Die Farberscheinungsattribute und damit auch die wahrgenommene Farbe können sich auf Grund verschiedener Effekte ändern. Einige werden im Folgenden genannt:

Stevens-Effekt

Der Stevens-Effekt beschreibt die Zunahme des wahrgenommenen Kontrastes bei steigender Umgebungshelligkeit [76]. So scheinen helle Stellen heller und dunkle Stellen dunkler zu sein, als messbar ist. Abbildung 3.4 verdeutlicht dies.

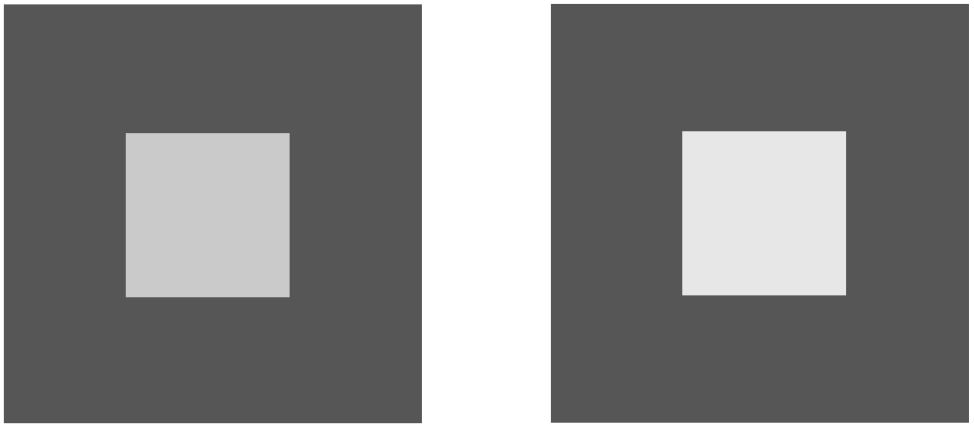


Abbildung 3.4: Stevens-Effekt

Hunt-Effekt

Steigt die Leuchtdichte einer Fläche an, steigert sich auch die Sättigung der entsprechenden Farbe [29]. Die Szene in Abbildung 3.5 auf der nächsten Seite hat eine erheblich höhere Leuchtdichte als in Abbildung 3.6. Dies bewirkt, dass vor allem die Blumen auf der linken Seite gesättigter erscheinen.

Abbildung 3.5: Hunt-Effekt - hohe Leuchtdichte, Bildquelle:[24]

Abbildung 3.6: Hunt-Effekt - niedrige Leuchtdichte, Bildquelle:[24]

Helmholtz-Kohlrausch-Effekt

Auch der Umkehrschluss aus dem Hunt-Effekt ist zu beobachten [16]: Wird die Sättigung einer Farbe erhöht, steigt auch die wahrgenommene Helligkeit, obwohl die Leuchtdichte der entsprechenden Stelle konstant bleibt. Die Stärke dieses Effekts ist abhängig von der jeweiligen Farbe.

Auch hier sollen zwei Bilder der Veranschaulichung dienen. Abbildung 3.7 zeigt ein Bild mit erhöhter Sättigung, Abbildung 3.8 ein Bild mit normaler Sättigung. Wiederum an den Blumen links ist gut erkennbar, wie sich der Helligkeitseindruck verändert.

Abbildung 3.7: Helmholtz-Kohlrausch-Effekt - erhöhte Sättigung, Bildquelle:[24]

Abbildung 3.8: Helmholtz-Kohlrausch-Effekt - niedrige Sättigung, Bildquelle:[24]

Bezold-Brücke-Effekt

Ändert sich die Leuchtdichte einer Fläche, verschiebt sich der wahrgenommene Farbton [61].

Dieser Effekt ist abhängig von der Empfindlichkeit der Photorezeptoren. Bei sehr hellen oder sehr dunklen Szenen können diese weniger Farben differenzieren. Daher wird in diesen Situationen eine Änderung des Farbtons wahrgenommen. Bei durchschnittlicher Beleuchtung tritt dieser Effekt dagegen nicht auf.

3.3.3 Farbtiefe

Ein wesentliches Kriterium für die Wiedergabequalität digitaler Bilder stellt die verwendete Farbtiefe dar. Dieser Begriff bezeichnet das Maß für die Menge an Abstufungen, die ein Farbwert annehmen kann [60]. Die Farbtiefe ist technisch bedingt und ergibt sich durch den Speicherplatz, der pro Farbkanal eines Pixels zur Verfügung steht. Dieser wird durch das gewählte Dateiformat oder die Verarbeitungskette vorgegeben.

Als Beispiel soll ein Schwarzweiß-Bild mit einer Farbtiefe von einem Bit dienen. In diesem Fall hat jedes Pixel einen Farbkanal, dessen Wert genau zwei Werte annehmen kann, nämlich 1 oder 0. Es können also zwei verschiedene Abstufungen pro Farbkanal wiedergegeben werden, hier schwarz und weiß. Dies bedeutet auch, dass keine Grautöne gespeichert werden können, die Wiedergabequalität ist also sehr stark begrenzt.

Für jedes Bit n, welches zusätzlich zur Verfügung steht, verdoppelt sich die Anzahl an möglichen Abstufungen x. Es gilt $x = 2^n$. Je größer die Farbtiefe, umso größer ist die Wiedergabequalität.

In den meisten Fällen werden Bilder mit einer Farbtiefe von 8 Bit abgespeichert, was 256 Abstufungen entspricht. Bei einem Schwarzweiß-Bild können so neben Schwarz und Weiß also noch 254 weitere Grautöne gespeichert werden.

Auch wenn dies eine wesentliche Steigerung im Vergleich zum aufgeführten Beispiel ist, gibt es viele Fälle, in denen auch diese Menge nicht ausreichend ist, um ein Bild adäquat zum Inhalt darzustellen. Sind im Bild sowohl sehr helle, als auch sehr dunkle Stellen sichtbar, muss der Abstand der 256 Werte entsprechend groß gewählt werden, damit der niedrigste und größte Wert den tatsächlichen Helligkeiten entspricht. Dadurch sind jedoch sehr feine Unterschiede zwischen den einzelnen Grautönen nicht mehr darstellbar. Üblicherweise wird zudem der umgekehrte Weg gewählt, sodass die Abstände der Grautöne bei verschiedenen Bildern gleich bleiben. In diesen Fällen wird an den oberen und unteren Darstellungsgrenzen der Farbwert beschnitten. Größere bzw. kleinere Werte werden jeweils als Weiß bzw. Schwarz dargestellt.

In solchen extremen Fällen ist die Verwendung eines Dateiformats erforderlich, welches 16 oder sogar 32 Bit pro Farbkanal zur Verfügung stellt. Eine gute Übersicht solcher Formate findet man bei Held [28].

3.4 Rendering

Der Begriff Rendering stammt aus der Computergrafik. Er bezeichnet die Synthese von Bildern. Durch die schnelle Entwicklung der Computer in den vergangenen Jahrzehnten

hat sich die Erzeugung von virtuellen Bildern und Bildsequenzen immer weiter etabliert. In vielen Fällen kann dadurch bereits ein fotorealistischer Eindruck erzielt werden.

Da 3D-Rendering als Basis dieser Arbeit dient, wird im Folgenden ein kurzer Überblick über den Rendering-Vorgang gegeben. Eine ausführlichere Beschreibung liefert z.B. [81].

Beim 3D-Rendering werden Objekte in einem virtuellen Raum zu einer Szene zusammengefügt. Durch Projektion werden deren 3D-Koordinaten dann auf einer zweidimensionalen Ebene abgebildet.

Diese Objekte können auf unterschiedliche Art und Weise repräsentiert werden. Eine Möglichkeit, die auch in dieser Thesis Verwendung findet, stellt die Erstellung von Polygonen dar. Dabei wird die Oberfläche jedes einzelnen Objekts durch Dreiecke (engl. Triangles) angenähert, die zusammen ein Netz (engl. Mesh) bilden. Ein Eckpunkt eines solchen Dreiecks wird als Vertex bezeichnet. Die Menge dieser Vertices stellen die 3D-Koordinaten dar, die für die Projektion benötigt werden.

Eine andere Möglicheit zur Repräsentation von Objekten sind sogenannte NURBS bzw. Non-Uniform Rational B-Splines. Dies sind Funktionen, mit denen Kurven und Flächen beschrieben werden können, durch deren Kombination die Oberfläche eines Objekts angenähert werden kann. NURBS eignen sich besonders, um organische Objekte zu repräsentieren. Allerdings ist die Ergänzung von Details aufwendig, da sich NURBS meist auf große Flächen beziehen, die in diesem Fall im Ganzen verändert werden müssen.

Die Projektion alleine reicht noch nicht aus, um eine tatsächlich darstellbare Grafik zu erzeugen. Bei den Objekten in der Szene handelt es sich um rein geometrische Daten, die keine Informationen über Pixelwerte liefern. Um diesen Formen ein Aussehen zu verleihen, werden zu jedem Objekt bestimmte Materialeigenschaften gespeichert. Sie geben Auskunft über Farbe und Struktur der Objektoberfläche.

Zusätzlich zu den Objekten werden virtuelle Lichtquellen in der Szene gespeichert. Sie sorgen wie reale Lichtquellen für die Ausleuchtung des abgebildeten Inhalts. So wird abhängig von den Lichtquellen und den Materialeigenschaften das Aussehen der Objekte beim Rendering beeinflusst. Optische Effekte wie Reflexion, Streuung oder Transparenz wären ohne virtuelle Lichtquellen nicht umsetzbar.

In der Regel befindet sich neben dem eigentlichen Inhalt ein zusätzliches Objekt in der Szene, welches als virtuelle Kamera fungiert. Es bildet das Zentrum der Projektion und dient so analog zu einer realen Kamera zur Definition der Perspektive. Beim Renderingprozess wird zunächst die Projektion der Koordinaten in Abhängigkeit von den Einstellungen der Kamera durchgeführt. Dabei werden passend zu der jeweiligen Perspektive auch die Verdeckungen der Objekte untereinander ermittelt.

Über zusätzliche Programme, die Shader, werden anschließend die korrespondierenden Pixelwerte berechnet. Das Resultat ist dann darstellbar.

3.4.1 Shader

Shader bilden eine Gruppe von Programmen, durch welche die Oberflächen von Objekten in einer Szene und damit zusammenhängende Effekte berechnet werden können.

Abhängig von der verwendeten Hard- und Software stehen beim Rendering verschiedene Shadertypen zur Verfügung. Weit verbreitet ist die Einteilung in Vertex Shader und Fragment Shader.

Vertex Shader

Wie der Name bereits andeutet, dienen Vertex Shader in erster Linie dazu, die Vertices eines Objekts zu verändern. Dadurch kann die Form eines Objektes und letztendlich auch dessen Beleuchtung verändert werden. Aus den Vertex-Koordinaten werden über Rasterisierung Fragments berechnet, die dann an den Fragment Shader übergeben werden können.

Fragment Shader

Ein Fragment Shader berechnet die Färbung dieser Fragments. Fragments sind die kleinsten Einheiten auf der Oberfläche eines Objektes und entsprechen derjenigen Fläche, die von einem korrespondierenden Pixel der Kamera erfasst wird. In der Regel werden Oberflächen, die der Kamera abgewandt sind, vor der Rasterisierung durch den Culling-Vorgang entfernt, sodass sie nicht vom Fragment Shader berücksichtigt werden. Ein Fragment Shader kann auch dem Kamera- bzw. dem Display-Objekt zugewiesen werden. In diesem Fall entsprechen die Fragments den Pixeln des Ausgabefensters.

Soll die Oberfläche eines Objekts nicht einfarbig, sondern gemustert dargstellt werden, kommt eine sogenannte Textur zum Einsatz. Dabei handelt es sich um ein zweidimensionales Bild, das dem Fragment Shader als zusätzlicher Input übergeben wird. Abhängig von der Position des Fragments bekommt dieses den Wert des korrespondierenden Pixels der Textur zugewiesen.

Oft sind Objekte nur im Hintergrund und gegebenenfalls unscharf in einer Szene zu sehen. In diesem Fall ergibt die Verwendung einer hoch aufgelösten Textur keinen Sinn, da deren Details nicht erkennbar wären. Um weniger Arbeitsspeicher zu benötigen, kann man auf kleinere Versionen der Textur zurückgreifen, die sogenannten Mipmaps. Diese werden durch Tiefpassfilterung der Textur erzeugt und liegen in mehreren Abstufungen, den Mipmap-Levels, vor. Der Renderer bestimmt automatisch, welche Stufe bei einer bestimmten Objektgröße verwendet wird.

Zu beachten ist, dass ein Fragment Shader zwar der gesamten Oberfläche zugewiesen wird, aber immer nur auf ein Fragment gleichzeitig zugreift. Abhängigkeiten unter den Fragments können daher nur begrenzt umgesetzt werden (vgl. Kapitel 7 auf Seite 39).

Neben diesen beiden Shadertypen existieren beispielsweise noch Geometry Shader, durch die u.a. komplette Objekte zur Szene hinzugefügt werden können, oder Tessellation Shader, über die einzelne Flächen nochmals unterteilt werden können.

Eine besondere Rolle spielen zudem die Compute Shader. Sie können beliebige Aufgaben lösen, die auch über den Grafikbereich hinaus gehen. Dadurch ist es möglich,

allgemeine Berechnungen über den Einsatz der Grafikkarte parallelisieren und damit beschleunigen zu können.

3.4.2 OpenGL

Standardmäßig werden Programme auf dem Rechner von der CPU ausgeführt. Sollen einzelne Prozesse davon abweichend auf der Grafikkarte berechnet werden, braucht man eine Schnittstelle, um Kommunikation zwischen CPU und GPU zu ermöglichen.

Eine solche Schnittstelle ist die Open Graphics Library oder OpenGL [80]. Sie ist plattformunabhängig angelegt und basiert auf C++. Es gibt jedoch auch Erweiterungen für andere Programmiersprachen. Der Quellcode dieser Arbeit wurde beispielsweise mit Java erstellt, wo OpenGL über die Light Weight Java Gaming Library (kurz „LWJGL“) eingebunden werden kann [44].

Bei OpenGL handelt es sich folgerichtig nicht um eine eigenständige Programmiersprache. Vielmehr wird der Funktionsumfang der als Basis dienenden Programmiersprache durch die Verwendung von OpenGL erweitert.

Zu dieser erweiterten Funktionalität gehört vor allem der Datentransfer zwischen Grafikkarte und CPU. Aber auch spezielle, auf 3D-Grafik ausgelegte Objekte und mathematische Funktionen sind in OpenGL spezifiziert.

Auch OpenGL ermöglicht die Verwendung von Shadern, sodass alle oben genannten Shadertypen umsetzbar sind. Die Programmierung der Shader erfolgt allerdings in GLSL (s.u.).

Für den Rendering-Prozess werden in OpenGL sogenannte FrameBuffer verwendet. Das sind Speicherstellen, in denen die Daten einer Szene sowie das Ergebnis des Renderings gespeichert werden. Der Zugriff erfolgt über FrameBufferObjects, auch „FBOs“ genannt. Wenn kein anderes FBO definiert wurde, benutzt OpenGL stattdessen standardmäßig das Ausgabefenster. Die Rendering-Ergebnisse werden so direkt auf dem Bildschirm ausgegeben.

Bei mehreren definierten FBOs kann der Speicherort des Renderings, das Rendertarget, vom Speicherort der Szene abweichen. Dadurch ist es möglich, das gerenderte Bild separat abzuspeichern und als zweite Textur für einen erneuten Rendering-Prozess zu verwenden. Dieser Vorgang wird als „Render-to-texture“ bezeichnet.

GLSL

GLSL steht für OpenGL Shading Language [80]. Sie dient als eigene Programmiersprache für die Shader, die von OpenGL angesteuert werden. Der Struktur nach ist GLSL stark an C angelehnt. So können Funktionen und Structures definiert werden, und die meisten Standard-Datentypen wie *Float*, *Integer*, *Byte*, *Boolean*, *String* und *Character* werden unterstützt.

Eine Besonderheit sind sogenannte *Sampler*, mit denen auf Texturen zugegriffen werden kann.

GLSL bietet zudem einige vorgefertigte Funktionen, die grundlegende mathematische Operationen ausführen. Dies sind z.B. Winkelfunktionen wie *sin()* und *cos()*, die Funktion *pow()* zur Potenzierung oder *dot()* zur Berechnung des Skalarprodukts. Eine vollständige

Auflistung aller Funktionen und Datentypen, die in GLSL unterstützt werden, liefert die GLSL-Spezifikation als Teil der Dokumentation von OpenGL.

Die Kommunikation zwischen OpenGL und GLSL funktioniert über sogenannte *uniform*-Variablen. Wird diesen von OpenGL aus ein Wert zugewiesen, kann ein Shader zur Laufzeit auf diesen Wert zugreifen. In GLSL kann dieser Wert so zwar ausgelesen, aber nicht verändert werden.

4 High Dynamic Range Imaging

Im vorherigen Kapitel wurde bereits auf den Dynamikumfang und dessen Bedeutung für die Erzeugung von Bildern eingegangen. Bei herkömmlichen bildgebenden Medien besteht ein enormer Unterschied zwischen dem Dynamikumfang, den das Medium darstellen kann, und demjenigen, den das Auge erfassen kann. Letzterer ist fast immer deutlich größer und kann durch den Adaptionsvorgang zusätzlich vergrößert werden. Dadurch entsteht ein wahrnehmbarer Unterschied zwischen dem Aussehen einer realen Szene und einem Abbild dieser Szene.

Abhilfe schafft in diesen Fällen „High Dynamic Range Imaging“ oder kurz „HDRI“. HDRI ist ein Oberbegriff für mehrere Verfahren, mit denen Bilder mit einem deutlich größeren Dynamikumfang als acht Blendenstufen erzeugt werden können.

4.1 HDR-Fotografie

4.1.1 HDR-Kamera

Der Stand der Technik ist aktuell so weit fortgeschritten, dass es bereits seit einiger Zeit Kameras gibt, die einen deutlich größeren Dynamikumfang erfassen können als herkömmliche Geräte. Ein Beispiel hierfür ist die S-Serie von Leica, deren aktuelles Modell „Type 006“ einen Dynamikumfang von 12 Blenden aufnehmen kann und mit einer Quantisierung von 16 Bit abspeichert [42]. Auch wenn dies noch weit hinter der Leistung des menschlichen Auges zurückbleibt, kann hier zumindest von Medium Dynamic Range (MDR) gesprochen werden.

Die Firma Spheron bietet mit der SpheroCam HDR [74] eine Spezialkamera an, mit der HDR-Panoramen erstellt werden können. Diese Bilder haben einen Dynamikumfang von bis zu 26 Blendenstufen und werden mit einer 32-Bit-Quantisierung in speziellen HDR-Formaten abgespeichert. Je nach Auflösung und gewünschtem Dynamikumfang kann eine solche Aufnahme jedoch bis zu mehreren Minuten dauern.

4.1.2 HDR-Rendering

Anstelle der Fotografie kann zur Erzeugung von rein virtuellen HDR-Bildern auch sogenanntes HDR-Rendering verwendet werden. HDR-Rendering unterscheidet sich vom herkömmlichen Rendering durch eine deutlich erhöhte Rechenpräzision bzw. Farbtiefe. So wird im Allgemeinen während der gesamten Berechnung auf 32-Bit-Werte zurückgegriffen. Hierdurch werden ein höherer Dynamikumfang und somit fotorealistische Beleuchtungsszenarien realisiert.

Die höhere Präzision bringt jedoch deutliche Einbußen in der Rechenzeit mit sich, was den Einsatz von HDR-Rendering in Echtzeitanwendungen erschwert. Nichtsdestotrotz ist dieses Verfahren zum jetzigen Zeitpunkt deutlich weiter entwickelt als die HDR-Fotografie. So ist es aktuell beispielsweise nur sehr eingeschränkt möglich, Video-Sequenzen mit hohem Dynamikumfang aufzuzeichnen (vgl. Abschnitt 4.4 auf Seite 27).

HDR-Rendering bildet daher auch die Grundlage für die in dieser Arbeit getesteten Verfahren.

IBL: Ein Sonderfall des HDR-Renderings stellt eine Methode namens Image Based Lighting dar (kurz: IBL) [13]. Hier wird ein fotografiertes HDR-Panorama dazu verwendet, eine virtuelle Szene auszuleuchten.

Das Bild wird auf die Oberfläche einer Kugel projiziert, welche die komplette Szene umhüllt. Diese sog. Lightprobe wird bei der Bildberechnung als Lichtquelle verwendet, welche die Szene abhängig vom Inhalt des HDR-Bildes beleuchtet.

4.2 HDR-Rekonstruktion

Auch mit herkömmlichen Kameras können HDR-Bilder aufgenommen werden, unter der Voraussetzung, dass die Blende manuell eingestellt werden kann. Im Unterschied zur Verwendung von HDR-Kameras wird das HDR-Bild hier jedoch nicht direkt aufgenommen, sondern aus einer Serie von Bildern rekonstruiert [66].

Dazu wird eine Belichtungsserie der Szene angefertigt, also mehrere Aufnahmen der Szene nacheinander, bei denen die Belichtung um jeweils eine Blendenstufe verändert wird. Es entstehen auf diese Weise sowohl Bilder, in denen die dunklen Stellen der Szene richtig belichtet sind, als auch solche, bei denen die hellen Stellen korrekt belichtet sind. Mit einem geeigneten Algorithmus können diese Bilder dann zu einem HDR-Bild zusammengefügt werden.

Aufgrund des zeitlichen Versatzes, der zwischen den einzelnen Aufnahmen liegt, kann dieses Verfahren nur bei statischen Szenen und statischer Kamera angewandt werden. Andernfalls entstünden sogenannte Geisterbilder der Objekte, welche sich bewegt haben (vgl. „Geisterbilder“, s.u.).

Für die Rekonstruktion des HDR-Bildes werden die bei der Aufnahme aufgetretenen linearen Beleuchtungswerte aus den Pixelwerten der RAW-Bilder bestimmt. Letztere sind in der Regel jedoch nicht linear, da jeder Sensor und somit jede Kamera unterschiedlich auf einfallendes Licht reagiert. Eine direkte Herleitung würde also das Ergebnis verfälschen.

Um die RAW-Bilder zu linearisieren, ist es demzufolge notwendig, das Verhalten des jeweiligen Sensors zu kennen. Dieses wird durch die Reaktionskurve der Kamera beschrieben, die jedoch von den meisten Kameraherstellern nicht veröffentlicht wird. Im folgenden Unterabschnitt wird daher ein Verfahren beschrieben, um die Reaktionskurve selbst zu bestimmen.

4.2.1 Reaktionskurve der Kamera

Grundlage für die Bestimmung der Reaktionskurve stellt ebenfalls eine Serie von Bildern des gleichen Motivs mit unterschiedlicher Belichtung dar. In der weiteren Vorgehensweise unterscheiden sich die einzelnen Algorithmen deutlich.

Als Beispiel dient hier die Variante von Mann und Picard [46]:

- Zunächst wird ein Pixel an der Stelle (x_0, y_0) des dunkelsten Bildes ausgewählt und dessen Wert p_0 betrachtet. Dieser entsteht durch die Anwendung der Reaktionskurve g auf eine undefinierte Lichtmenge q_0, daher gilt $p_0 = g(q_0)$.

- Als nächstes wird der Pixelwert p_1 an der gleichen Stelle im zweiten Bild gesucht und mit dem ersten Bild in Relation gesetzt: $p_1 = k * p_0 = g(k * q_0)$.

- Der selbe Vorgang wird für das dritte Bild wiederholt. Es gilt: $p_2 = k * p_1 = g(k^2 * q_0)$. Auch für die weiteren Bilder kann diese Beziehung hergestellt werden, wobei sich der Wert p_n des Bildes n aus $p_n = g(k^n * q_0)$ ergibt.

- Diese Kette wird nun für alle Pixel durchgeführt und aus den jeweiligen Pixelpaaren ein sogenannter Range-Range-Plot gebildet. Dieser gibt das Verhältnis von $g(k*q_0)$ zu $g(q_0)$ in Abhängigkeit von q_0 wieder.

- Aus diesem Range-Range-Plot kann dann die Reaktionskurve hergeleitet werden, wobei jedoch einige Annahmen im Vorhinein getroffen werden müssen. So wird angenommen, dass g semimonoton[1] ist und einen Nullpunkt bei $q_0 = 0$ hat, wobei Sensorrauschen außer Acht gelassen wird. Zudem ist bekannt, dass eine Filmkurve der Exponentialfunktion $f(q) = \alpha + \beta q^\gamma$ entspricht: Die aus einer Belichtung E resultierende Dichte D des Filmmaterials kann beschrieben werden als $D(logE) = \alpha + \beta \log E^\gamma$. Sie wird daher oft auch als D-logE-Kurve bezeichnet. Da $q_0 = 0$ ist, fällt α weg. Daraus folgt für die Beziehung der Pixelpaare zueinander

  $$p_{n+1} = k^\gamma p_n. \tag{4.1}$$

  Mann et al. schlagen zur Ermittlung von γ die Verwendung von Regression über alle Punkte p vor [46].

Neben diesem Algorithmus existieren weitere, zum Teil deutlich aufwendigere Verfahren wie [50], [69] und [12].

Die Berechnung der Reaktionskurve muss nur einmal pro Kamera erfolgen. Ist sie bekannt, kann sie als Grundlage für alle mit dieser Kamera fotografierten HDR-Bilder verwendet werden.

Mit gegebener Reaktionskurve g können mit deren Inversen g^{-1} die nichtlinearen Pixel $p_n(x, y)$ linearisiert werden. Die Pixelwerte werden anschließend über die Belichtungszeit Δt normiert. Um die letztendlichen Beleuchtungswerte $E'_e(x, y)$ zu erhalten,

[1] Eine Funktion $f(x)$ ist semimonoton, wenn sie mit bx steigt. Für den Spezialfall $b = 1$ ist die Funktion monoton [79].

werden die Pixelwerte gemittelt und über eine Wichtungsfunktion $w(p_n(x,y))$ abhängig von der Beleuchtung gewichtet [63]: Über- oder unterbelichtete Pixel fließen so weniger in das Ergebnis mit ein als korrekt belichtete.

Der ganze Prozess lässt sich über die Gleichung (4.2) darstellen:

$$E'_e(x,y) = \frac{\sum_{n=1}^{N} w(p_n(x,y)) \frac{g^{-1}(p_n(x,y))}{\Delta t}}{\sum_{n=1}^{N} w(p_n(x,y))}. \tag{4.2}$$

4.2.2 Rauschunterdrückung

Ein Problem des Verfahrens zur Rekonstruktion ist, dass das Bildrauschen, welches grundsätzlich in allen Bildern vorhanden ist, bei den unterbelichteten Aufnahmen künstlich verstärkt wird. Dies ist gerade bei Nachtaufnahmen kritisch, bei denen der Rauschpegel generell sehr hoch ist. Es ist daher ratsam, bei der Rekonstruktion des HDR-Bildes Rauschunterdrückung anzuwenden. Dies kann z.B. durch eine einfache Durchschnittsbildung der Bilder geschehen, wodurch verrauschte Pixel dann eliminiert werden. Reinhard et al. zeigen in [63], wie Rauschunterdrückung mit der HDR-Rekonstruktion verbunden werden kann.

4.2.3 Geisterbilder

Geisterbilder (engl. „ghosting“) treten dann auf, wenn sich die Objekte der fotografierten Szene während des Aufnahmeprozesses bewegt haben. Dies kann auch bei eigentlich statischen Objekten geschehen, wenn sehr dünne oder sehr leichte Gegenstände beispielsweise durch einen Luftzug ihre Position verändern.

Auch eine Bewegung der Kamera kann zu Geisterbildern führen. Dieses Risiko kann zwar durch die Verwendung eines Stativs beseitigt werden, zur Sicherheit sollte aber zudem ein Fernauslöser benutzt werden. Selbst die leichte Berührung des Auslöseknopfs kann sonst genug Bewegung verursachen, um erkennbare Geisterbilder entstehen zu lassen.

Durch Bewegungsschätzung [32] oder ein Varianz-basiertes Verfahren [66] lassen sich Geisterbilder wieder entfernen.

4.3 Pseudo-HDR

Neben diesen Aufnahmeverfahren gibt es eine Reihe von Bildbearbeitungstechniken, die einen ähnlichen Eindruck erzeugen sollen, wie herkömmliche HDR-Fotografie. Bei diesen Techniken wird jedoch kein echtes HDR-Bild erzeugt, welches anschließend durch Tone Mapping dargestellt wird. Somit handelt es sich nicht um HDR-Verfahren im eigentlichen Sinne.

4.3.1 Dodging-and-burning

Mit Dodging-and-burning wird eine relativ alte Vorgehensweise aus der analogen Fotografie beschrieben, bei der einzelne Teile des Bildes besonders über- ($\triangleq$ burning)

oder unterbelichtet (≙ dodging) werden [1]. Dazu wird während der Belichtung die entsprechende Stelle entweder abgedeckt und dadurch geringer belichtet, oder der Rest des Bildes wird abgedeckt, wodurch die Stelle stärker belichtet wird.

Mit diesem Verfahren kann der tatsächliche Dynamikumfang des Bildes deutlich erhöht werden. Durch die bewusste Wahl der verdeckten Bildbereiche sowie der begrenzten Genauigkeit der Schablonen entspricht das Ergebnis jedoch keinem „echten“ HDR-Bild.

4.3.2 Exposure-Fusion

Eine ähnliche Methode wie Dodging-and-burning, die in der Digitalfotografie Anwendung findet, stellt die sogenannte Exposure-Fusion oder Exposure-Merging-Methode dar [63]. Aus einer Belichtungsreihe werden verschiedene Bereiche der einzelnen Bilder ausgewählt und freigestellt, die in das Endergebnis mit einfließen sollen. Mit einer Bildbearbeitungssoftware werden diese Bildausschnitte dann wieder zu einem vollständigen Bild zusammengefügt bzw. fusioniert.

Die Funktionsweise des Verfahrens ist fast identisch mit Dodging-and-burning. Da die Auswahl der verschiedenen Bildbereiche jedoch erst nach der Belichtung stattfindet, benötigt man unterschiedliche Bilder als Quellen für die Highlights und die Shadows. Auch bei den Ergebnissen der Exposure-Fusion handelt es sich nicht um HDR-Bilder im eigentlichen Sinn.

4.4 HDR-Video

Die Möglichkeiten, HDR-Videos aufzuzeichnen, sind zurzeit noch deutlich eingeschränkter als im Standbildbereich. Allerdings sind auch hier einige wichtige technische Erneuerungen zu vermerken.

Die prinzipiell am weitesten entwickelte Möglichkeit für HDR-Videosequenzen ist das HDR-Rendering. Zwar stellt die deutlich höhere Datenrate durch die erforderliche 32-Bit Farbtiefe eine Einschränkung dar, die gerade im Bereich der interaktiven Anwendungen bisher für Schwierigkeiten sorgte. Jedoch gibt es mittlerweile erste Renderer, welche HDR-Bildsequenzen in Echtzeit erzeugen können, wie z.B. OctaneVR [55]. Bei Anwendungen ohne Echtzeitanforderung wird HDR-Rendering schon seit längerer Zeit eingesetzt [15].

HDR-Rendering ist jedoch auf virtuelle Szenen beschränkt. Bei realen Motiven ist die Aufnahme schwieriger. HDR-Rekonstruktion kann i.d.R. nicht verwendet werden, da hierzu mehrere Aufnahmen der Szene erforderlich sind. Es gibt allerdings mit MagicLantern eine Software, welche einen ähnlichen Ansatz verfolgt [40]. Das für Canon-Kameras entwickelte Programm schaltet während der Aufnahme zwischen zwei aufeinander folgenden Bildern die ISO-Einstellung um, also die Empfindlichkeit der Kamera. Es werden so pro Sekunde 12,5 Bilder überbelichtet und 12,5 Bilder unterbelichtet, welche in der Postproduktion miteinander kombiniert werden. Dies erhöht zwar den Dynamikumfang, jedoch nicht in dem Maße, als dass es als „echtes“ HDR zu bezeichnen wäre. Zudem sind durch den zeitlichen Versatz zwischen unter- und überbelichtetem Bild deutliche Ghosting-Artefakte zu erwarten.

Daneben existieren mehrere Varianten, die zwei Kameras gleichzeitig einsetzen. Diese werden entweder über ein Spiegelrig auf die gleiche Position gebracht [23] oder möglichst eng beieinander montiert, wobei im Nachhinein eine Korrektur der Perspektive durchgeführt werden muss [25]. Auch auf diese Weise kann der Dynamikumfang deutlich erhöht werden. Die resultierende Anzahl an aufnehmbaren Blendenstufen hängt dabei von den verwendeten Kameras ab.

Ähnlich wie im Standbildbereich gibt es auch verschiedene Videokameras, welche grundsätzlich einen höheren Dynamikumfang aufzeichnen können als herkömmliche Modelle. So erzielt beispielsweise die Pocket Cinema Camera von BlackMagic bis zu 13 Blendenstufen [7]. Auch bei diesem Modell ist jedoch eher von Medium Dynamic Range zu sprechen als von HDR.

Die erste „echte“ HDR-Video-Kamera wurde ebenfalls von der Firma Spheron entwickelt und 2010 auf der Siggraph vorgestellt. Die SpheronVR HDRv sei laut Hersteller dazu in der Lage, bis zu 50 fps mit einer Auflösung von 2.336 mal 1.752 Pixeln aufzunehmen [73] und erfasse dabei einen Dynamikumfang von ca. 22 Blenden. Zum aktuellen Zeitpunkt ist diese Kamera jedoch noch nicht käuflich zu erwerben. Eine weitere HDR-Kamera wurde 2011 von Contrast Optical Design & Engineering vorgestellt. Die AMP-Kamera umfasst einen Dynamikumfang von 17 Blenden [3].

4.5 HDR-Monitore

Herkömmliche Monitore unterliegen ähnlichen Einschränkungen wie Kameras. Durch die begrenzte Farbtiefe sowie die eingeschränkte Ansteuerbarkeit der verwendeten Lampen können sie den vollen Kontrastumfang eines HDR-Bildes nicht darstellen. Auch hier findet daher in der Regel Clipping statt, sodass entweder die hellen oder die dunklen Stellen im Bild abgeschnitten werden.

Seit einiger Zeit exisitieren allerdings HDR-Monitore, die einen wesentlich größeren Dynamikumfang abbilden können. Bereits 2005 stellte beispielsweise die Firma BrightSide den DR37-P vor, einen HDR-TV Prototypen. Er arbeitet mit einer Farbtiefe von 16 Bit pro Farbkanal und kann Helligkeitsunterschiede von bis zu fünf Größenordnungen bzw. 16 Blendenstufen darstellen [68].

5 Tone Mapping

Die Auswahl an HDR-Monitoren auf dem Markt ist, wie oben beschrieben, zum jetzigen Zeitpunkt sehr begrenzt. Gleichzeitig sind die Vorteile der Verwendung von HDR-Bildern aber offensichtlich. Um solche Bilder auch an herkömmlichen Monitoren mit eingeschränktem Dynamikumfang darstellen zu können, müssen die großen Pixelwerte des Bildes über ein mathematisches Verfahren in den geringeren darstellbaren Wertebereich umgerechnet werden. Solche Verfahren heißen Tone Mapping Operatoren (kurz TMO), Tone Mapping Verfahren oder schlicht Tone Mapper.

Grundsätzlich handelt es sich beim Tone Mapping um eine Luminanzkompression, da diese maßgeblich für die Größe der Pixelwerte verantwortlich ist. In den meisten Fällen wird die Luminanz daher als erster Schritt des Verfahrens aus den Pixelwerten ermittelt, komprimiert und die Farbkanäle anschließend entsprechend angepasst. Es ist jedoch auch möglich, die Kompression direkt auf die Farbkanäle anzuwenden. Dies kann abhängig vom verwendeten Farbraum jedoch zu Abweichungen in der Wirkung des Bildes führen, wenn sich die jeweiligen Farbwerte nicht proportional zur Helligkeit verhalten.

Die existierenden Verfahren unterscheiden sich durch die Genauigkeit der Vorgehensweise, durch die Zielanwendung und letztendlich auch durch den verwendeten mathematischen Ansatz. Das Ergebnis soll i.d.R. entweder möglichst „schön“aussehen oder möglichst realistisch, wobei in letzterem Fall entweder ein HDR-Monitor oder eine reale Szene als Referenz dient.

Der denkbar einfachste Fall wäre eine lineare Kompression, bei der die Pixelwerte durch einen festen Faktor dividiert werden. Dem Ergebnis gingen jedoch die meisten Details verloren und es sähe daher weder visuell ansprechend noch real aus.

Tone Mapping Operatoren lassen sich in vier Gruppen aufteilen. Man unterscheidet im räumlichen Bereich zwischen globalen Operatoren, bei denen jedes Pixel nach dem gleichen Verfahren komprimiert wird, und lokalen Operatoren, bei denen die nähere Umgebung des Pixels in die Berechnung des Kompressionsgrades mit einfließt.

Globale Operatoren haben den Vorteil, dass sie im Allgemeinen wesentlich schneller arbeiten als lokale Operatoren. Gleichzeitig sind die Ergebnisse jedoch oft ungenauer. Es soll nicht unerwähnt bleiben, dass einige Verfahren sich sowohl als globale als auch als lokale Version umsetzen lassen, wie z.B. der fotografische Operator von Reinhard [67].

Die dritte Gruppe bilden die frequenzbasierten Operatoren. Hier wird das Bild in den Frequenzbereich transferiert, um hohe von den niedrigen Frequenzen zu unterscheiden und unterschiedlich komprimieren zu können.

Eine weitere Gruppe stellen die gradientenbasierten Verfahren dar. Hier ist der Kompressionsgrad abhängig von der Stärke der Gradienten im Bild.

5.1 Globale Operatoren

Globale TMOs komprimieren alle Pixelwerte durch eine Formel, welche auf global ermittelten Parametern beruht. Diese Parameter werden pro Bild ermittelt und sind für alle Pixel des Bildes identisch. Die Lage des Pixels selbst bzw. seine Umgebung hat keinen Einfluss auf die Kompression.

Als globale Parameter dienen z.B. minimale und maximale Helligkeit, Durchschnittshelligkeit oder die Log-Average-Luminance, das geometrische Mittel der Helligkeiten.

Da diese Parameter nur einmal pro Bild ermittelt werden müssen, ist die Anzahl der notwendigen Berechnungen bei globalen Operatoren relativ gering. So ist die benötigte Rechenzeit ebenfalls deutlich niedriger als bei anderen Verfahren. Allerdings gehen in einigen Fällen durch die reine Verwendung globaler Parameter Details verloren.

Beispiel: Sigmoid-Kurve

Ein Beispiel für einen globalen TMO ist das Verfahren von Reinhard und Devlin [64], welches auf der Reaktionsweise von Photorezeptoren basiert. Diese komprimieren die Lichtmenge im Auge nach der Gleichung

$$V = \frac{I}{I + \sigma(I_a)} \cdot V_{max} \tag{5.1}$$

mit I als Intensität des einfallenden Lichts, V_{max} als maximal möglicher Sehreiz und $\sigma(I_a)$ als Semisaturationskonstante. Es gilt ferner:

$$\sigma(I_a) = (f \cdot I_a)^m. \tag{5.2}$$

Dieser Wert gibt an, wie stark die Photorezeptorzelle an eine bestimmte Adaptionslichtstärke I_a angepasst ist, wobei f und m die Intensität bzw. den Kontrast steuern.

Reinhard und Devlin schlagen vor, die Parameter f und m weiter anzupassen, um eine möglichst automatische Berechnung oder eine zumindest intuitive Beeinflussung des Ergebnisses gewährleisten zu können. So lässt sich m annähern durch

$$m = 0.3 + 0.7 \cdot k^{1.4}. \tag{5.3}$$

Der Parameter k entspricht hier dem sogenannten „key-value“, welcher angibt, ob ein Bild generell eher dunkel oder eher hell ist. Er lässt sich wiederum berechnen über

$$k = (L_{max} - L_{av})/(L_{max} - L_{min}) \tag{5.4}$$

und ist damit vollständig automatisierbar. Drückt man f als Exponentialfunktion der Form

$$f = exp(-f') \tag{5.5}$$

aus, kann f' nun als Nutzerparameter verwendet werden. Dadurch kann die Menge von sinnvollen Einstellwerten auf einen Bereich von -8 bis 8 beschränkt werden, wobei 0 einer neutralen Stellung entspricht (vgl. [64]).

Trägt man in einem Diagramm nun die Eingangsintensitäten gegen die Sehreize auf, erhält man einen S-förmigen Kurvenverlauf.

Beispiel: Histogramm-Anpassung

Ein weiterer globaler Algorithmus stellt das Verfahren von Larsson et al. [41] dar, welches die Luminanz des Bildes über eine Histogramm-Anpassung reduziert.

Zunächst wird das Bild so tiefpassgefiltert, dass bei gegebenem Bildwinkel der Durchmesser eines Pixels ungefähr einem Grad entspricht:

$$S = 2tan(\theta/2)/0.01745. \tag{5.6}$$

Hier ergibt sich die resultierende Bildbreite bzw. -höhe S in Pixeln aus dem vollständigen horizontalen bzw. vertikalen Sichtwinkel θ und dem Umrechnungsfaktor 0.01745 von Grad nach Bogenminuten.

Aus den so erhaltenen Pixelwerten wird zunächst jeweils die Luminanz berechnet und diese anschließend logarithmiert. Diese Werte werden dann in ein Histogramm eingeteilt. Larson et al. unterteilen den gesamten Wertebereich in 100 gleichgroße Abschnitte.

Zur Anpassung des Kontrasts wird nun eine Funktion verwendet, welche die obere akzeptable Grenze pro Histogrammeinheit darstellt. Wird diese Grenze überschritten, werden die entsprechenden Werte einfach abgeschnitten. Auf diese Weise wird der Kontrast jedoch nur verändert, falls er tatsächlich nicht von herkömmlichen Monitoren darstellbar ist. Unnötige Luminanzreduktionen werden vermieden.

Als Grenzfunktion dient der sogenannte „just noticable difference“(kurz JND) $\triangle L_t(L_a)$, der kleinste wahrnehmbare Unterschied bei einer Adaptionshelligkeit L_a. Dies ist der durch das Auge gerade noch wahrnehmbare Helligkeitsunterschied bei einer bestimmten Umgebungshelligkeit. Zur Kontrastanpassung wird der JND bei Displayhelligkeit ins Verhältnis gesetzt zum JND der Beleuchtungssituation der echten Szene. Es ergibt sich:

$$\frac{\delta L_d}{\delta L_w} \leq \frac{\triangle L_t(L_d)}{\triangle L_t(L_w)}. \tag{5.7}$$

Der Kontrast darf nicht größer sein als das Verhältnis der JNDs.

Des Weiteren sieht das Verfahren weitere Anpassungen an die menschliche Sichtweise vor, wie Blendung bei sehr hellen Stellen und verringerte Farbempfindlichkeit sowie verringerte Schärfe bei dunklen Bildbereichen.

5.2 Lokale Operatoren

Bei lokalen Tone Mapping Operatoren haben neben den jeweiligen Pixelwerten und globalen Bildparametern auch sogenannte lokale Parameter Einfluss auf die Kompression. Das bedeutet, dass auch Informationen über die nähere Umgebung des jeweiligen Pixels benötigt bzw. berechnet werden. Man geht davon aus, dass auch das Auge sich nicht an das gesamte Bild, sondern an die jeweilige Region anpasst. Daher werden Pixel in hellen Regionen anders komprimiert als solche, die in dunklen Regionen liegen.

Insgesamt ist es somit möglich, Bilder mit einem deutlich höheren Dynamikumfang zufriedenstellend zu komprimieren, sodass deutlich mehr Details erhalten bleiben.

Gleichzeitig sind lokale Operatoren jedoch anfälliger für Artefakte wie z.B. Halos, also Rändern an Grenzen verschiedener Regionen. Zudem ist die Berechnung in der Regel komplexer und erfordert somit mehr Rechenzeit.

Beispiel : Multiscale Retinex

Der Multiscale Retinex Operator von Rahman et al. [62] greift die Retinex-Theorie von Land auf (siehe [39] und [38]). Sie besagt, dass der Mensch keine absoluten „lightness“-Werte für ein ganzes Bild wahrnimmt, sondern relative, von den einzelnen Bildregionen abhängige Werte.

Diese Werte, Retinex genannt, lassen sich berechnen, indem man das Verhältnis zwischen dem jeweiligen Pixelwert und den gemittelten Werten der umgebenden Pixel berechnet.

Hierfür werden die Pixelwerte zunächst logarithmiert. Den Wert der Umgebung erhält man durch Faltung des logarithmierten Pixelwertes mit einer Filterfunktion F(x,y). Der Retinex-Wert ergibt sich dann durch:

$$R_i(x,y) = \log I_i(x,y) - \log\left[F(x,y) * I_i(x,y)\right]. \tag{5.8}$$

Für die Filterfunktion gilt

$$F(x,y) = Ke^{-(x^2+y^2)/c^2}, \tag{5.9}$$

wobei c der Filtergröße entspricht.

Bei der Multiscale-Version wird das Bild jedoch in mehrere Spektralbänder aufgeteilt, für die jeweils die Retinex-Funktion berechnet wird. Das Gesamtbild erhält man durch gewichtete Addition der einzelnen Retinex-Werte:

$$R_{M_i}(x,y) = \sum_{n=1}^{N} w_n R_{n_i}(x,y). \tag{5.10}$$

Hier ist N die Anzahl der Skalen bzw. Frequenzbänder, R_{n_i} der Retinex-Wert des jeweiligen Bandes und w_n das entsprechende Gewicht. Durch geeignete Wahl von w_n kann die Luminanz komprimiert werden.

Es hat sich gezeigt, dass für die meisten Fälle identische Gewichte für alle Skalen gewählt werden können. Bekommt z.B. das erste Frequenzband deutlich mehr Gewicht als die übrigen, entspricht das zwar der größten Kontrastreduktion, es entstehen jedoch gleichzeitig starke Artefakte.

Beispiel: Fotografischer Tone Mapping Operator

Ein weit verbreiteter lokaler TMO ist der fotografische Algorithmus von Reinhard et al. [67]. Dieser ist mit dem Ziel entwickelt worden, visuell möglichst ansprechende Bilder zu produzieren und gleicht die Ergebnisse an die typische Erscheinung von Fotografien an.

Der Algorithmus existiert sowohl als globale, als auch als lokale Variante. Die ersten Schritte sind zunächst gleich: Für den Tone Mapper wird als erstes ein „key-value“ ermittelt, der auch bei dem bereits vorgestellten Sigmoid-Algoithmus verwendet wird (Abschnitt 5.1 auf Seite 30). Er gibt gleichzeitig den Wert an, auf den das mittlere Grau des Bildes im Resultat abgebildet wird.

Der Key kann vom Nutzer festgelegt oder automatisch berechnet werden. Die Abbildung erfolgt dann über:

$$L_{mapped} = \frac{key}{L_{av}} \cdot L. \tag{5.11}$$

Die Luminanz-Kompression erfolgt dann über eine Sättigungskurve mit der Form:

$$L_{display} = \frac{L_{mapped} \cdot (1 + \frac{L_{mapped}}{L_{white}^2})}{1 + L_{mapped}} \tag{5.12}$$

mit L_{white} als Luminanz des Zielweißpunktes. Die Klammer im Zähler sorgt dafür, dass Highlights kontrolliert ausbrennen, was den Autoren nach einem natürlicheren Eindruck entspricht. Dieses Merkmal ist auch bei Fotografien zu beobachten.

Diese Funktion kann von einer globalen auf eine lokale Variante gebracht werden, indem eine digitale Variante von „Dodging-and-burning“ (siehe Abschnitt 4.3.1 auf Seite 26) implementiert wird.

Dazu wird das L_{mapped} im Nenner durch ein $L_{surround}$ ersetzt, welches mit Hilfe einer weichgezeichneten Version des Originalbildes ermittelt wird. Der Algorithmus sieht vor, dass die dazu verwendete Filtergröße schrittweise optimiert wird, wozu eine Art Marr-Hildreth-Operator genutzt wird.

Filteroptimierung: Der Marr-Hildreth-Operator ist ein Kantendetektor [47]. Er wird durch die Differenz zweier Laplace-Operatoren mit unterschiedlich großer Varianz erzeugt. Eine mathematische Annäherung erhält man durch die Verwendung zweier Gauß-Filter, was auch als Difference of Gaussian (kurz DoG) bezeichnet wird.

Der verwendete Operator baut auf einem DoG-Operator von [8] auf. Dazu wird folgende Center-Surround-Funktion aufgestellt:

$$V(x, y, s) = \frac{V_1(x, y, s) - V_2(x, y, s)}{2^{\phi}\alpha/s^2 + V_1(x, y, s)}. \tag{5.13}$$

Hier sind α der bereits erwähnte Key-Wert und ϕ ein Schärfeparameter. Die Parameter V_1 und V_2 sind zwei Filterantworten, die jeweils durch eine Faltung des Bildes mit einem Gaußfilter entstehen. Die Varianz s des Gaußfilters von V_2 ist dabei etwas größer als von V_1. Das hat zur Folge, dass große Kontraständerungen durch Kanten in einer größeren Differenz und damit einem größeren V resultieren.

Für den lokalen TMO ist davon auszugehen, dass bei optimaler Filtergröße $s_{optimal}$ V_2 als Surround-Filter einen größtmöglichen Bereich um das Zentrum abdeckt, in dem keine großen Kontraständerungen auftreten (vgl. [67]). Wird die Varianz von V_1 und V_2 schrittweise erhöht, erreicht man die optimale Größe, sobald V einen Grenzwert ε überschreitet. Es gilt:

$$|V(x, y, s_{optimal})| < \varepsilon. \tag{5.14}$$

5.3 Frequenzbasierte Operatoren

Frequenzbasierte Operatoren teilen das Bild in hohe und niedrige Frequenzen auf. Während die großen Luminanzwerte, die für den hohen Dynamikumfang verantwortlich

sind, meist von den niedrigen Frequenzen repräsentiert werden, entsprechen hohe Frequenzen Details im Bild, die möglichst erhalten bleiben sollen. Die Kompression findet daher entweder ausschließlich auf tiefen Frequenzen statt oder zumindest dort deutlich stärker als auf den Details.

Beispiel: Bilateraler Filter

Als bilateralen Filter bezeichnet man einen Filter, welcher aus zwei gefalteten Gaußfiltern besteht [17]. Dabei arbeitet einer der Gaußfilter in der Ortsdomäne, der andere in der Intensitätsdomäne. Durch den zusätzlichen Intensitätsfilter werden Umgebungspixel, deren Intensität stark von derjenigen des Zielpixels abweicht, als Ausreißer betrachtet und fließen so weniger stark in die Berechnung des Ergebnisses mit ein [82]. Dies führt dazu, dass Stellen im Bild mit großem Kontrast erhalten bleiben, wie z.B. Kanten.

Durand und Dorsey nutzen den bilateralen Filter, um ein Base-Layer mit niedrigen Frequenzen zu erhalten. Eine Division des Ursprungsbildes durch das Base-Layer liefert dann ein Detail-Layer mit den hohen Frequenzen. Das Tone Mapping findet anschließend nur auf dem Base-Layer statt, wodurch die Feinheiten im Detail-Layer erhalten bleiben. Damit umgehen die Autoren ein Hauptproblem der globalen Operatoren, durch die feine Details oft verloren gehen.

Für die eigentliche Dynamikkompression werden die Luminanzwerte des Base-Layers zunächst logarithmiert. Über den tatsächlich im Bild vorhandenen Kontrast und den gewünschten Kontrast wird dann ein Skalierungsfaktor ermittelt, mit dem die Log-Werte multipliziert werden. Es handelt sich damit um eine frequenzabhängige Abwandlung eines globalen Operators nach Tumblin und Turk [83].

Beschleunigung: Die Faltung des Bildes mit den Gaußfiltern ist sehr rechenintensiv. Durand und Dorsey implementieren daher eine *stückweise lineare Annäherung* an den bilateralen Filter, um die Berechnung zu beschleunigen. Dazu werden die möglichen Intensitätswerte I zunächst diskretisiert, d.h. auf eine maximal mögliche Anzahl $NBSEGMENTS$ verschiedener Werte begrenzt. Für jeden Wert I_s des Signals werden nun die beiden nächsten Werte aus $NBSEGMENTS$ ausgewählt, hier benannt als i_1 und i_2. Die Filterantwort für I_s ergibt sich dann aus der linearen Interpolation der Filterantworten für i_1 und i_2.

Der Vorteil dieser Vorgehensweise ist, dass i_1 und i_2 durch Verwendung der Fast-Fourier-Transformation berechnet werden können. Dies ist bei direkter Berechnung von I_s nicht möglich, da der Filter dann signalabhängig ist.

Eine weitere Steigerung der Performance kann erzielt werden, indem das Bild vor der Filterung mit einem einfachen Tiefpass gefiltert wird. Dadurch wird der präziser arbeitende bilaterale Filter nur auf einem deutlich kleineren Bild ausgeführt und benötigt so weniger Rechenzeit. Es ist leicht zu erkennen, dass diese Art der Beschleunigung zu einem Verlust an Qualität führen muss. Ab welcher Filtergröße des ersten Tiefpasses dieser bemerkbar ist, wurde von Durand und Dorsey [17] nicht näher untersucht. Diese Tiefpassfilterung wird über *Nearest-Neighbor-Interpolation* umgesetzt. Dabei erhält das neue Pixel den Wert desjenigen Pixels, welches räumlich am nächsten liegt. Bildlich gesprochen wird also jedes zweite Pixel ausgelassen.

5.4 Gradientenbasierte Operatoren

Die Grundlage von gradientenbasierten TMOs bildet die Tatsache, dass die Helligkeit einer Szene aus tatsächlicher Beleuchtung L einerseits und Reflexion R aus der Umgebung andererseits besteht. Letztere spielt für den hohen Dynamikumfang jedoch kaum eine Rolle, sodass man für die Kompression nur L berücksichtigen muss. L und R lassen sich aber kaum von einander trennen. Die Entwickler gradientenbasierter Operatoren (z.B. Fattal et al. [22]) berufen sich auf eine Studie von [77], aus der sich schließen lasse, dass R sich deutlich schneller ändere als L, weswegen man L durch einen Tiefpassfilter ermitteln könne.

Legt man auf das Bild nun einen Tiefpassfilter, so bewirkt jede große Änderung in der Luminanz auch große Werte für die Gradienten des Bildes, vorausgesetzt die Stärke des Tiefpassfilters ist richtig gewählt. Durch Reduktion dieser Gradienten lässt sich das Bild komprimieren.

Beispiel: Verfahren von Fattal

Ein gradientenbasiertes Verfahren stellt der Algorithmus von Fattal et al. dar [22]. Hier wird versucht, durch einen Schwellenwert Gradienten von Details und Kanten zu trennen. Diese können dann unterschiedlich stark komprimiert werden.

Da es in den meisten Bildern Kanten unterschiedlicher Stärke gibt, ist es notwendig, die Gradienten in mehreren Subbildern zu suchen. Jedes Subbild entspricht dabei der Tiefpassfilterung der vorhergehenden, was durch die Verwendung einer Gauß-Pyramide verwirklicht wird (s. Abschnitt 7.7 auf Seite 76).

Den jeweiligen Gradienten des Subbildes k an der Stelle (x, y) erhält man durch

$$\nabla H_k = (\frac{H_k(x+1,y) - H_k(x-1,y)}{2^{k+1}}, \frac{H_k(x,y+1) - H_k(x,y-1)}{2^{k+1}}). \tag{5.15}$$

Die eigentliche Kompression erfolgt dann über folgende Gleichung:

$$\varphi_k(x,y) = \frac{\alpha}{\|\nabla H_k(x,y)\|} \cdot (\frac{\|\nabla H_k(x,y)\|}{\alpha})^{\beta}. \tag{5.16}$$

Der Parameter α dient hier als Schwellenwert für die Kompression. Nur Gradienten, die größer sind als α, werden so komprimiert. Über β ist die Kompressionsstärke regelbar.

Der so berechnete Wert gilt jedoch nur für das jeweilige Subbild. Um die tatsächlichen Skalierungsfaktoren zu ermitteln, werden die φ-Werte von Subbild zu Subbild übertragen, bis das Ursprungsbild als letzte Stufe erreicht ist. Dies geschieht über:

$$\Phi_d(x,y) = \varphi_d(x,y) \tag{5.17}$$
$$\Phi_k(x,y) = L(\Phi_{k+1})(x,y)\varphi_k(x,y) \tag{5.18}$$
$$\Phi(x,y) = \Phi_0(x,y). \tag{5.19}$$

d ist die höchste Stufe, 0 die niedrigste bzw. das Originalbild. $Phi_0(x,y)$ stellt somit den finalen Skalierungswert dar. L ist ein Upsampling-Operator, der auf linearer Interpolation beruht.

6 Gamut Mapping

Die CIE definiert in ihren „Guidelines for the Evaluation of Gamut Mapping Algorithms“ den Begriff Colour Gamut als „eine Menge von Farben, welche durch ein gegebenes Farbreproduktionsmedium [...] unter bestimmten Betrachtungsverhältnissen erzeugt werden können“. Weiterhin sei Gamut „ein Volumen im Farbraum“[1] [11].

Ein Bildschirm mit einer Farbauflösung von 8 Bit pro Farbkanal kann insgesamt 256 * 256 * 256 verschiedene Farben darstellen. Diese Menge entspricht somit seinem Gamut, welcher sich im RGB-Farbraum befindet.

Da der Gamut spezifisch für jedes Wiedergabemedium ist, unterscheidet sich in den meisten Fällen die Farbdarstellung von Wiedergabemedium zu Wiedergabemedium. Um diese Unterschiede auszugleichen, muss ein Verfahren gefunden werden, was jede Farbe des ersten Gamuts einer Farbe des zweiten Gamuts zuweist. Dieses Verfahren wird Gamut Mapping genannt [11].

Der Umfang und die Funktionsweise des Gamut Mapping Algorithmus ist abhängig vom angestrebten Ziel. Da der Gamut des Reproduktionsmediums kleiner als der Ursprungsgamut sein kann, muss und kann die Erscheinung der Farben durch Gamut Mapping nicht immer bewahrt bleiben [51].

6.1 CAM

Ein CAM oder Color Appearance Model ist ein mathematisches Modell, um die Farberscheinung eines Bildes zu bestimmen. CAMs werden in vielen Gamut-Mapping-Verfahren eingesetzt, wenn der ursprüngliche Farbeindruck beim Transfer zwischen verschiedenen Farbräumen erhalten bleiben soll. Das mit Abstand am meisten verbreitete CAM ist das CIECAM02-Modell der CIE, welches mittlerweile als Quasi-Industriestandard gilt.

6.1.1 Chromatische Adaption

Chromatische Adaption ist die mathematische Umsetzung des visuellen Adaptionsprozesses an wechselnde Lichtarten (vgl. Abschnitt 3.2 auf Seite 7). Sie kann ebenfalls eingesetzt werden, um den Farbeindruck eines Bildes zu erhalten. Daher ist auch im CIECAM02-Verfahren die chromatische Adaption vorgesehen.

[1] Übersetzung des Autors

Im Folgenden wird von Farbwerten in XYZ-Koordinaten als Ursprung ausgegangen. Für die chromatische Adaption werden diese zunächst in einen geschärften RGB-Farbraum transformiert, der genauer an die Empfindlichkeitskurven der Photorezeptoren angepasst ist, als andere RGB-Farbräume [43].

Die entsprechende Farbraumtransformation wird CIECAT02-Transformation genannt. Zugrunde liegt folgende Transformationsmatrix:

$$M_{CAT02} = \begin{Bmatrix} 0.7328 & 0.4296 & -0.1624 \\ -0.7036 & 1.6975 & 0.0061 \\ 0.0030 & 0.0136 & 0.9834 \end{Bmatrix}. \tag{6.1}$$

Für die Adaption der Werte muss zunächst der Adaptionsgrad D berechnet werden. Dieser gibt an, wie stark das Auge bereits an die neue Lichtart angepasst ist. D berechnet sich über

$$D = F[1 - \frac{1}{3.6} e^{\frac{-L_A - 42}{9.2}}]. \tag{6.2}$$

Der Exponent L_A ist die Helligkeit, an welche die Farbwerte adaptiert werden sollen. F ist ein Vorfaktor mit einer ähnlichen Funktion wie der des „key"-Wertes des fotografischen TMOs. Für eine durchschnittlich helle Umgebung hat F den Wert 1.

Der adaptierte Farbwert lässt sich dann über folgende Formel berechnen:

$$R_c = [(Y_W \frac{D}{R_W}) + (1 - D)]R. \tag{6.3}$$

Hier wurde beispielhaft der Wert des L-Kanals eingesetzt. G_c und B_c können analog berechnet werden.

Durch Verwendung der invertierten Transformationsmatrix werden die adaptierten Werte wieder in XYZ-Koordinaten umgerechnet.

6.1.2 Farberscheinungsattribute

Ein weiterer Bestandteil eines Gamut Mapping Algorithmus kann die Anpassung der Farberscheinungsattribute sein.

Diese werden durch Transformation in einen geeigneten Farbraum wie IPT oder ICh ermittelt. In CIECAM02 werden die XYZ-Koordinaten dazu in den HPE-Farbraum transferiert. Vorgesehen sind die Berechnung des Farbtons h, der relativen und absoluten Helligkeit J und Q, der relativen und absoluten Buntheit C bzw. M sowie der Sättigung s.

6.2 Operatoren

Auf den Farberscheinungsattributen aufbauend beginnt das eigentliche Gamut Mapping. Es gibt dabei eine große Anzahl von Möglichkeiten, die Attribute zu beeinflussen (vgl. Morovic[51]).

Grundsätzlich unterscheidet man zwischen Reduktionsverfahren, die von einem großen Gamut in einen kleineren transformieren sollen, und Expansionsverfahren, welche in umgekehrter Weise arbeiten. Letztere spielen für diese Thesis keine Rolle.

6.2.1 Color-by-Color-Reduktion

Eine Möglichkeit der Reduktion besteht in der direkten Kompression der jeweiligen Farbe unabhängig von ihrer Umgebung. Diese Vorgehensweise wird als Color-by-Color-Reduktion bezeichnet (siehe ebenfalls Morovic[51]). Auch hier gibt es wiederum verschiedene Herangehensweisen.

So ist es beispielsweise möglich, die Farbattribute sequentiell entlang eines Pfades in den kleineren Gamut zu projizieren. Eine naheliegende Kompression des „lightness"-Attributs sieht wie folgt aus [9]:

$$D_L = G_{D_{max}L} - (G_{S_{max}L} - S_L)\frac{G_{D_{max}L} - G_{D_{min}L}}{G_{S_{max}L} - G_{S_{min}L}}. \tag{6.4}$$

Hier symbolisiert S_L den Wert der Helligkeit. $G_{D_{max}L}$ bzw. $G_{D_{min}L}$ sind die maximale bzw. minimale Helligkeit des Zielgamuts, die Variablen mit G_S stellen die analogen Werte des Ursprungsgamuts dar. Das Ergebnis D_L schließlich ist der Wert, auf den die Helligkeit komprimiert wird.

Ähnlich dazu kann dann ebenfalls die Chrominanz adaptiert werden, wie es z.B. beim LLIN-Operator [71] der Fall ist. Auch der Farbwinkel hue kann entsprechend verändert werden.

In neueren Verfahren (z.B. Ito et al. [33] und Neumann et al.[52]) ist es jedoch üblich, die Attribute nicht mehr entlang eines einzigen Pfades zu projizieren, sondern mehrere Projektionszentren zu verwenden, um die Form des Ursprungsgamuts besser zu bewahren.

Auch werden zum Teil nicht alle Farbwerte komprimiert. Gibt es eine Menge von Farbwerten, die sowohl im Ursprungs- als auch im Zielfarbwert existieren, wird ein innerer Bereich dieser Region unverändert übernommen. Die Werte, welche den Zielgamut überschreiten, werden dann im äußeren Bereich abgebildet (vgl. Ito et al. [33] und MacDonald [45]).

6.2.2 Spatial-Gamut-Reduktion

Color-by-Color-Reduktion hat jedoch den Nachteil, dass einige Details im Bild verloren gehen können. Diese Vorgehensweise stellt im Prinzip ein globales Tone Mapping mit den entsprechenden Nachteilen dar.

Aus diesem Grund existieren auch beim Gamut Mapping lokale Verfahren, die unter dem Begriff Spatial-Gamut-Reduktion zusammengefasst werden. Dabei gibt es ebenfalls verschiedene Ansätze.

Ähnlich wie beim bilateralen Filter besteht z.B. die Möglichkeit, das Bild in einen niedrigfrequenten und einen hochfrequenten Teil aufzuteilen. Die Kompression findet dann nur auf den niedrigen Frequenzen statt. Die Details in den hohen Frequenzen bleiben unverändert und können dem Bild nach dem Gamut Mapping wieder hinzugefügt werden (s. Meyer und Barth [49]).

Daneben gibt es auch ein Verfahren, was auf der Retinex-Theorie beruht [48]. Analog zum entsprechenden Tone Mapping Operator wird das Bild in einzelne Frequenzbänder aufgeteilt. Diese werden dann jeweils über Color-by-Color-Reduktion komprimiert und in Abhängigkeit von ihrer Gewichtung wieder zusammengefügt.

7 Farbgetreues Tone Mapping

Für die Erstellung eines farbgetreuen Tone Mapping Operators wurden einige bestehende Verfahren implementiert, die in den folgenden Abschnitten beschrieben werden. Alle Verfahren beinhalten einen oder mehrere Bestandteile von Gamut Mapping Verfahren, welche den Farberscheinungseffekten entgegen wirken sollen. Alle Verfahren wurden als GLSL-Shader umgesetzt. Als Renderer dient ein auf OpenGL basierendes Programm, welches am Institut für Medien und Phototechnik der Fachhochschule Köln entwickelt wurde. Dieses ist echtzeitfähig und basiert auf IBL.

Die Verfahren werden miteinander verglichen, um die beste Variante zu ermitteln. Maßgeblich für die Echtzeitfähigkeit ist eine Gesamtlaufzeit von unter 16 Millisekunden. Die Wiedergabe der Farben bzw. die Farberscheinung wird subjektiv ermittelt. Die Auswertung ist in Kapitel 8 auf Seite 89 zu finden.

Für die Testbilder wurden die verwendeten 3D-Szenen mit HDR-Lightprobes beleuchtet. Diese entstammen der Light Probe Image Gallery von Paul Debevec [14].

7.1 L_{av}, L_{min} und L_{max}

Für fast alle der eingesetzten Operatoren werden Informationen über die Helligkeitsverhältnisse im Bild benötigt. Die meisten Tone Mapper greifen auf die Log-Average Luminance bzw. das geometrische Mittel aller Luminanz-Werte im Bild (kurz: L_{av}) zurück. Es dient als eine grundlegende Annahme darüber, ob eine Szene eher dunkel oder eher hell ist. Darauf basierend passt der Algorithmus die Kompression der Helligkeitswerte entsprechend an. Einige Operatoren benötigen zudem die minimale und maximale Helligkeit im Bild (L_{min} bzw. L_{max}).

Zur Berechnung dieser Werte bedarf es Informationen über alle Bildpunkte gleichzeitig. Fragment Shader allerdings sind darauf ausgelegt, jeweils nur ein Fragment bzw. Pixel zu bearbeiten und so über die parallele Verarbeitung einen enormen Performance-Gewinn zu erzielen. Es ist zwar möglich, den Zugriff auf andere Pixel im Fragment Shader zu programmieren, damit ist der Performance-Gewinn jedoch hinfällig. Zudem werden dann pro Pixel alle anderen Pixel ebenfalls aufgerufen, was in einer immensen Zahl an überflüssigen Operationen und schließlich dem Absturz des Programms endet.

Daher liegt es nahe, die Luminanzwerte zurück zur CPU zu transferieren, dort L_{av}, L_{min} und L_{max} zu berechnen und anschließend über geeignete Variablen dem Shader wieder zur Verfügung zu stellen.

Die Umsetzung des geometrischen Mittels in Java wird mit Hilfe einer geschachtelten For-Schleife berechnet. Hierzu werden die ln der Luminanzwerte erst addiert

und dann außerhalb der Schleife normiert. Die exp-Funktion liefert anschließend die Durchschnittshelligkeit.

Innerhalb der gleichen For-Schleife wird ebenfalls das Minimum des aktuellen Bildes gesucht, indem der jeweilige Luminanzwert mit dem aktuellen Minimum verglichen wird. Die Berechnung des Maximums erfolgt analog.

Implementierung

Abbildung 7.1 verdeutlicht die Berechnung von L_{max}, L_{min} und L_{av}.

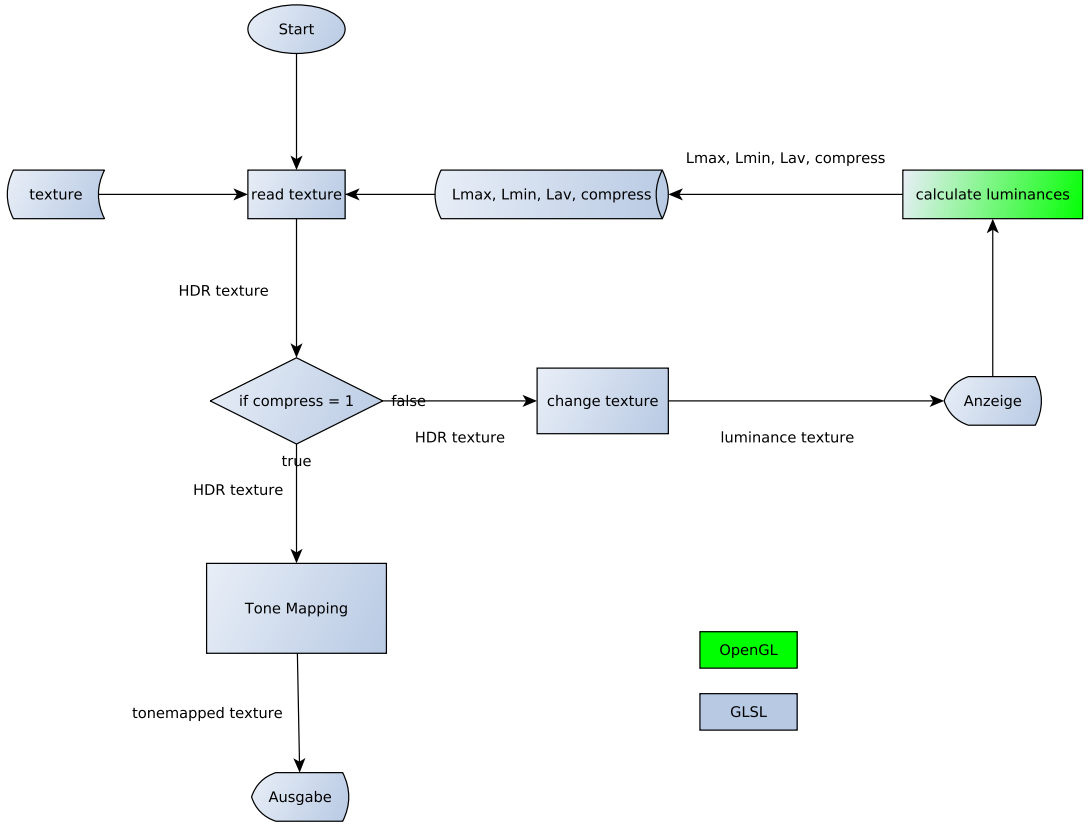


Abbildung 7.1: Pseudocode Luminanzberechnung Shader

Während des Renderings wird der aktive Shader zum ersten Mal aufgerufen. Dort wird die HDR-Textur eingelesen und der Wert der Variable „compress“ überprüft. Diese hat standardmäßig den Wert 0. In diesem Fall wird kein Tone Mapping durchgeführt, sondern die Textur verkleinert, wobei in den Farbkanälen des ausgegebenen Bildes die Werte für L_{max}, L_{min} und L_{av} des jeweiligen Pixels gespeichert werden:

```
1  /**
   * Auszug zur Berechnung der Luminanz-Textur im Shader
3  */
...
5 if (0 == compress){
  // Get new coords, only take every second pixel
7  vec2 coords = gl_FragCoord.xy*2;
```

```
  // Get average value
  vec4 top = mix(texelFetch( uRadiance, ivec2(coords), 0 ),
    texelFetch( uRadiance, ivec2(coords)+ivec2(0,1), 0 ), 0.5);
  vec4 bottom = mix(texelFetch( uRadiance, ivec2(coords)+ivec2(1,0),
    0 ), texelFetch( uRadiance, ivec2(coords)+ivec2(1,1), 0 ), 0.5);
  vec4 average = mix(top,bottom,0.5);

  // Get minimal value
  top = min(texelFetch( uRadiance, ivec2(coords), 0 ), texelFetch(
    uRadiance, ivec2(coords)+ivec2(0,1), 0 ));
  bottom = min(texelFetch( uRadiance, ivec2(coords)+ivec2(1,0), 0 ),
    texelFetch( uRadiance, ivec2(coords)+ivec2(1,1), 0 ));
  vec4 minimum = min(top,bottom);

  // Get maximal value
  top = max(texelFetch( uRadiance, ivec2(coords), 0 ), texelFetch(
    uRadiance, ivec2(coords)+ivec2(0,1), 0 ));
  bottom = max(texelFetch( uRadiance, ivec2(coords)+ivec2(1,0), 0 ),
    texelFetch( uRadiance, ivec2(coords)+ivec2(1,1), 0 ));
  vec4 maximum = max(top,bottom);

  // Set luminances as new color channels
  tonemapped.r = 0.2126 * average.r + 0.7152*average.g + 0.0722*
    average.b;
  tonemapped.g = 0.2126 * minimum.r + 0.7152*minimum.g + 0.0722*
    minimum.b;
  tonemapped.b = 0.2126 * maximum.r + 0.7152*maximum.g + 0.0722*
    maximum.b;
}
...
```

Diese Textur wird dann in OpenGL ausgelesen, wo über folgende Methode die absoluten Luminanzwerte ermittelt werden:

```
/**
* Methode zur Berechnung von Lmax, Lmin und Lav
*/
void calculateLuminances( ByteBuffer image){
  double sum = 0.0;
  for(int i =0; i< (height/2); i++){
    for(int j =0; j < (width/2); j+=4){

      // entspricht 4 Bytes pro Kanal pro Pixel
      int k = width * 4 * 4 * i + j;
      //Erster Kanal für Lav
      float tmp =(image.get(k)+image.get(k+1)+image.get(k+2)+image.
    get(k+3));
      //Zweiter Kanal für Lmin
      float min = (image.get(k+4)+image.get(k+5)+image.get(k+6)+image
    .get(k+7));
      //Dritter Kanal für Lmax
      float max =
      (image.get(k+8)+image.get(k+9)+image.get(k+10)+image.get(k+11))
    ;
      if(min < lmin){
```

```
19            lmin=min;}
          else if(max > lmax){
21            lmax = max;}
          if(tmp!=0){
23            // Addiere die Logarithmen
              sum += Math.log(tmp);
25        }
      }
27    }
    //get the average value and exponentiate
29    lav= (float) Math.exp((1.0/(width*height))*sum);
}
```

Abschließend werden diese Werte dem Shader über entsprechende Uniform-Variablen übergeben und der Wert von „compress“ ebenfalls über eine Uniform-Variable auf 1 gesetzt. Bei dem nun folgenden zweiten Durchlauf des Shaders wird das Tone Mapping auf der Original-Textur angewandt. Dieses Verfahren ist für alle implementierten Tone Mapping-Shader identisch.

Optimierung

Die Berechnung der Luminanzwerte kann über mehrere Shader-Passes optimiert werden. Das bedeutet, dass pro Durchlauf der Renderschleife nicht nur ein Fragment Shader ausgeführt wird, sondern mehrere nacheinander. So kann die Luminanzberechnung komplett auf die Grafikkarte verlegt werden und das Auslesen der Textur wird überflüssig.

Für diese Optimierung werden zunächst zwei neue Shader definiert. Kasten 7.1 zeigt einen Shader, der pro Pixel zunächst die Luminanz berechnet und diesen Wert dann in jedem Farbkanal des Ausgabepixels hinterlegt. Es entsteht ein Luminanzbild.

```
'''
2   Dieser Shader berechnet die Luminanz eines Pixels und speichert
      diesen Wert in jedem Farbkanal.
'''
4 in  vec2 vTexCoord;
  out vec4 FragColor;
6


8 layout( binding = 0 ) uniform sampler2D uRadiance;

10 void main(void)
  {
12
    #Get Pixel
14  vec4 color = texelFetch( uRadiance, ivec2(gl_FragCoord.xy), 0 );

16  #Calculate Luminance
    float lum = 0.2126*color.r + 0.7152*color.g + 0.0722*color.b;
18
    #Return luminance image
20  FragColor = vec4(lum,lum,lum,1);
  }
```

Listing 7.1: Preprocessing Shader

Der Shader in Kasten 7.2 greift nun dieses Luminanzbild wieder auf. Er fragt die Werte eines 4x4 großen Pixelblocks ab und berechnet aus den Farbkanälen den jeweiligen Maximal-, Minimal- und Log-Avarage-Wert. Diese fasst er in einem Ausgabepixel zusammen. Die Größe der Textur wird also gleichzeitig um den Faktor 16 reduziert.

```
'''
  Auszug aus dem Reduction Shader.
'''
void main(void) {
  vec4 pixels[16];
  #recalculate texture coords
  vec2 coords = gl_FragCoord.xy*4;

  #Get Pixels
  pixels[0] =  texelFetch( uRadiance, ivec2(coords), 0 );#bottom left
  pixels[1] =  texelFetch( uRadiance, ivec2(coords) + ivec2(0, 1), 0
   ); #bottom right

 #... etc. until...

  pixels[15] = texelFetch( uRadiance, ivec2(coords) + ivec2(3, 3), 0
   ); #top right

  #lmax - blue channel
  float blMax = max(max(pixels[0].b, pixels[1].b),  max(pixels[2].b,
   pixels[3].b)  );
  float brMax = max(max(pixels[5].b, pixels[8].b),  max(pixels[10].b,
    pixels[11].b));
  float tlMax = max(max(pixels[4].b, pixels[9].b),  max(pixels[7].b,
   pixels[13].b) );
  float trMax = max(max(pixels[6].b, pixels[14].b), max(pixels[12].b,
    pixels[15].b));
  float lmax = max(max(blMax, brMax), max(tlMax, trMax));

 #lmin - green channel
  float blMin = min(min(pixels[0].g, pixels[1].g),  min(pixels[2].g,
   pixels[3].g)  );
  float brMin = min(min(pixels[5].g, pixels[8].g),  min(pixels[10].g,
    pixels[11].g));
  float tlMin = min(min(pixels[4].g, pixels[9].g),  min(pixels[7].g,
   pixels[13].g) );
  float trMin = min(min(pixels[6].g, pixels[14].g), min(pixels[12].g,
    pixels[15].g));
  float lmin = min(min(blMin, brMin), min(tlMin, trMin));

  #lav - red channel
  float blAv = sqrt(pixels[0].r*pixels[0].r + pixels[1].r*pixels[1].r
      + pixels[2].r*pixels[2].r   + pixels[3].r*pixels[3].r  );
  float brAv = sqrt(pixels[5].r*pixels[5].r + pixels[8].r*pixels[8].r
      + pixels[10].r*pixels[10].r + pixels[11].r*pixels[11].r);
  float tlAv = sqrt(pixels[4].r*pixels[4].r + pixels[9].r*pixels[9].r
      + pixels[7].r*pixels[7].r   + pixels[13].r*pixels[13].r);
  float trAv = sqrt(pixels[6].r*pixels[6].r + pixels[14].r*pixels
   [14].r + pixels[12].r*pixels[12].r + pixels[15].r*pixels[15].r);
  float lav = sqrt(blAv*blAv + brAv*brAv + tlAv*tlAv + trAv*trAv);
```

```
37
    FragColor = vec4(lav, lmin, lmax, 1.0);
39 }
```

Listing 7.2: Reduction Shader

Über mehrere erneute Aufrufe dieses Shaders kann die Texturgröße weiter reduziert werden. So erhält man am Ende ein einziges Pixel, welches den Maximal-, Minimal- und den Log-Avarage-Wert des gesamten Bildes enthält [1].

Alle diese Aufrufe der Shader stellen jeweils einen eigenen Shaderpass dar. Diese Passes werden der Reihe nach in einer ArrayList gespeichert, um sie nach und nach abzuarbeiten. Die Erstellung dieser Liste wird in Kasten 7.3 veranschaulicht.

```
1  /**
     * Auszug aus der Main-Methode
3    */
   ArrayList<GLRenderPass> passList = new ArrayList<>();
5
   //Rendert das Bild
7  passList.add( new GLRenderPass(Rendermode.POSTPROCESS, null, "
       Tonemap_vs.glsl", "" , "Null_fs.glsl") );

9  //Rendert das normale Bild in FBO2 - speichere das Ergebnis in
       Textur2
   passList.add( new GLRenderPass(Rendermode.POSTPROCESS,
       preprocessCallback, "Tonemap_vs.glsl", "" , "Null_fs.glsl") );
11
   //Graustufenbild in Textur0
13 passList.add( new GLRenderPass(Rendermode.POSTPROCESS, null, "
       Tonemap_vs.glsl", "" , "Preprocess_fs.glsl") );

15 //Erzeuge Reduce-Passes
   int minDim = Math.min(Main.panel.getHeight(), Main.panel.getWidth());
17 int reductionFaktor = (int) Math.ceil(Math.log(minDim)/Math.log(4));

19 for(int i=0; i< (reductionFaktor); i++){
     //Reduziere Textur0
21   passList.add(new GLRenderPass(Rendermode.POSTPROCESS, null, "
       Tonemap_vs.glsl", "","Reduction_fs.glsl"));
   }
23
   //Speichere Textur0 in FBO 1, lade Ergebnis in Textur1
25 passList.add(new GLRenderPass(Rendermode.POSTPROCESS,
       changeTexCallback, "Tonemap_vs.glsl", "","Null_fs.glsl"));

27 //Render Input von Tex2 in Tex0
   passList.add(new GLRenderPass(Rendermode.POSTPROCESS, null, "
       Tonemap_vs.glsl", "","PassThrough_fs.glsl"));
```

Listing 7.3: Erstellung der Renderpasses

[1] Anm.: Dies gilt für quadratische Texturen. Bei ungleicher Seitenlänge kann die Anzahl der resultierenden Pixel größer sein.

Die Variable *passes* enthält zunächst eine leere Liste. Dieser wird nun zuerst ein Pass mit dem Preprocessing-Shader hinzugefügt. Die Anzahl der notwendigen Reduction-Shader-Passes hängt davon ab, wie oft sich die Textur verkleinern lässt. Wird der Shader zu oft aufgerufen, sind alle Pixel schwarz. Wird er zu selten aufgerufen, wird die Textur nicht weit genug reduziert. Maßgeblich ist hier die kleinste Seite der Textur, die in der Variablen *minDim* gespeichert wird. Pro Durchlauf des Shaders wird die Seitenlänge um den Faktor 4 verkleinert. Die Anzahl der möglichen Durchgänge ergibt sich also über den Logarithmus zur Basis 4.

Der so berechnete Reduktionsfaktor wird in der Variable *reductionFaktor* gespeichert. Über eine For-Schleife werden nun in Abhängigkeit von *reductionFaktor* die Reduction-Shader-Passes zur Liste hinzugefügt. Die Liste ist damit vollständig.

Für die weiteren Operationen ist es wichtig, dass die Luminanzwerte in einer zusätzlichen Textur gespeichert werden. Im anderen Fall ginge die Originaltextur verloren. Hierzu wird für jeden Shaderpass der entstandenen Liste ein sogenannter RenderCallback definiert. In diesem kann angegeben werden, was beim Aufruf des zugehörigen Shaders geschehen soll. Der in Kasten 7.4 dargestellte Callback wird verwendet, um das ursprüngliche, unveränderte Bild zunächst in einem zweiten FBO zu speichern.

```
preprocessCallback = new RenderCallback()
    {
        @Override
        public void onRender( Mat4 viewMatrix, Mat4 projMatrix,
    ArrayList<GLMesh> meshes, ArrayList<Material> materials, ArrayList
    <Mat4> transforms, GLTexture[] userdata, int resourceIndex)
        {
                // set OpenGL state, textures, swap framebuffers, etc.
    here
                // postprocess passes always get a fullscreen quad as
    input => meshes.get(0) is a screen filling quad

                GL30.glBindFramebuffer(GL30.GL_FRAMEBUFFER,
    framebuffer2ID);
                for( GLMesh mesh : meshes )
                        mesh.draw();

                GL30.glBindFramebuffer(GL30.GL_FRAMEBUFFER,0);
                GL13.glActiveTexture(GL13.GL_TEXTURE2);
                GL30.glGenerateMipmap(GL_TEXTURE_2D);
                glBindTexture(GL_TEXTURE_2D, mipmapTextureID);

        }
};
```

Listing 7.4: Preprocess Callback

Über *glBindFramebufferEXT()* wird vor dem Rendering das aktuelle FBO geändert. Die Rückgabewerte des Shaders werden jetzt in der Textur gespeichert, die an das FBO mit dem Index *framebuffer2ID* gebunden ist. Nach dem Rendering wird wieder das Ausgabefenster als Standard-FBO mit dem Index 0 aktiviert. Durch *glActiveTexture()* wird der dritte Texturslot des FBOs aktiviert. An diesen wird nun über *glBindTexture* die Textur gebunden, welche den Index *mipmapTextureID* besitzt.

Das aktive FBO besitzt jetzt also zwei gleiche Texturen. Über den Aufruf des Preprocessing-Shaders wird im ersten Slot dann die Luminanztextur erzeugt. Diese wird danach über den Reduction-Shader verkleinert. Hierfür ist kein zusätzlicher Callback notwendig. Nachdem die Textur ein letztes Mal verkleinert wurde, wird diese dem zweiten FBO zugewiesen und von dort aus über einen entsprechenden Callback in den zweiten Texturslot kopiert.

Über einen zusätzlichen Shader werden anschließend die Pixelwerte aus dem dritten wieder in den ersten Texturslot kopiert, damit sich dort wieder die Originaltextur befindet. Den Quellcode dieses PassThrough-Shaders ist in Kasten 7.5 dargestellt.

```
in  vec2 vTexCoord;
out vec4 FragColor;

layout( binding = 2 ) uniform sampler2D uRadiance;

#Uebertraegt die Textur ohne weitere Verarbeitung
void main(void)
{
    vec4 radiance   = texelFetch( uRadiance, ivec2(gl_FragCoord.xy),
   0 );
  FragColor = radiance;

  if( any(isnan(FragColor)) )
    FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
  if( any(isinf(FragColor)) )
    FragColor = vec4( 0.0, 1.0, 0.0, 1.0 );
}
```

Listing 7.5: Zugriff auf die zusätzliche Textur

Im Shader kann so über den Sampler auf die zweite Textur zugegriffen werden. Damit dies funktioniert, muss allerdings der Index, der über *layout(binding = 2)* definiert wird, mit dem Index des verwendeten Texturslots übereinstimmen. Ein Auslesen der Textur und die Übergabe der Werte über die uniform-Variablen ist so überflüssig.

Nach diesem Kopiervorgang verfügt das erste FBO über drei Texturen: Im ersten Slot befindet sich das ursprüngliche Bild, im zweiten die Luminanztextur und im dritten ein zweites Mal das ursprüngliche Bild. Dieser Texturslot wird jedoch nicht weiter benötigt.

7.2 Calibrated image appearance reproduction

Ziel dieses Algorithmus ist die realitätsgetreue Reproduktion eines Bildes unter bekannten Betrachtungsbedingungen [65]. Er wird im weiteren Verlauf der Thesis mit „CIAR“ abgekürzt.

Input Parameter

Das Originalbild muss in linearen, absoluten Werten vorliegen, idealerweise in XYZ-Koordinaten mit Y-Werten in $\frac{cd}{m^2}$. Dazu werden die normierten RGB-Werte zunächst in den XYZ-Farbraum transferiert. Zusätzlich werden die Log-Average Luminanz sowie der Weißpunkt der Hauptlichtquelle in XYZ-Werten benötigt. Der Weißpunkt wird unter Umständen angepasst, um Entsättigung zu vermeiden. Es folgt die Berechnung des Adaptionsgrades des visuellen Systems analog zu CIECAM02:

$$d = 1 - \frac{\exp((-L_{av} - 42)/92)}{3.6}. \tag{7.1}$$

Als maximale Helligkeit wird $\frac{9}{10}$ der hellsten Stelle im Bild angesetzt. Auf diese Weise wird sichergestellt, dass einzelne, sehr helle Stellen im Bild keinen Einfluss auf die weiteren Berechnungen haben, wodurch der Algorithmus robuster wird. Mit diesem Wert wird der zugehörige Weißpunkt berechnet.

Diese Parameter werden nun sowohl für das Display als auch für den Betrachtungsraum ermittelt. Sie sind jedoch sachbedingt vom Nutzer definiert, also nicht aus dem Bild berechenbar. Es ergibt sich:

- $L_{avDisplay} = 100 =$ Y-Wert des Display-Whitepoints
- $L_{avSurround} = 100 =$ Y-Wert des Surround-Whitepoints
- $L_{maxDisplay} = 5 * L_{avDisplay}$
- $L_{maxSurround} = 5 * L_{avSurround}$

Abhängig vom Betrachtungswinkel werden diese Parameter dann zu zusammengesetzten Parametern der Betrachtungsumgebung kombiniert, um weitere Berechnungen zu vereinfachen:

$$L_{avViewing} = \alpha * L_{avDisplay} + (1 - \alpha) * L_{avSurround} \tag{7.2}$$

$$L_{maxViewing} = \alpha * L_{maxDisplay} + (1 - \alpha) * L_{maxSurround} \tag{7.3}$$

Alle Werte werden anschließend in den LMS-Farbraum transformiert.

Pupil size

Mit der Pupille kann die Lichtmenge gesteuert werden, die auf die Netzhaut fällt. Die Größe der Pupille stellt hier eine Funktion in Abhängigkeit von L_{max} dar. Mit ihrer Hilfe können die Leuchtdichten in die Einheit Troland umgerechnet werden. Sie wird sowohl für die Betrachtungsumgebung als auch für die Szene berechnet.

Bleaching

Der Effekt des Bleachings wird als zweiter Faktor der obigen Berechnung hinzugefügt. Er dient dazu, eine Übersättigung der Nervenzellen zu simulieren, bei der die betrachtete Stelle weiß erscheint.

Photoreceptor response

Als Kompressionsfunktion dient eine Variante der Michaelis-Menten-Gleichung:

$$V_s = V_{max} * \frac{I}{I + \sigma_s * \frac{V_max}{f_s}}. \tag{7.4}$$

I ist dabei der Eingangsreiz der Photorezeptoren, V_{max} ist die Sättigungsgrenze bzw. der maximal mögliche Wert, den V_s annehmen kann. σ_s ist die Halbsättigungskonstante und stellt den Einfluss der Adaption dar, während f_s ein zusammengesetzter Faktor aus Bleaching und Pupillengröße darstellt. Diese Kompressionsfunktion wird jedoch etwas abgewandelt, um bei unterschiedlicher Betrachtungsumgebung den gleichen visuellen Eindruck zu erzielen.

Final mapping function

Es ergibt sich die finale Tone Mapping Funktion:

$$L = L_{max} * \frac{I}{I + \tau * L_{max}} \tag{7.5}$$

mit

$$\tau = \frac{\sigma_{scene}}{f_{scene}} * \frac{f_{viewing}}{\sigma_{viewing}}. \tag{7.6}$$

Implementierung

Der zugehörige Shader ist trotz der Parallelisierung fast identisch mit dem oben beschriebenen Verfahren. Allerdings werden alle genannten Schritte auf jeweils ein Pixel angewandt und nicht auf das gesamte Bild auf einmal.

Parameter: Die Lichtverhältnisse für die Szene, das Betrachtungsumfeld und den Monitor wurden in Anlehnung an die Matlab-Implementierung von [59] angenähert. Es ergeben sich die Werte in Kasten 7.6, Kasten 7.7 und Kasten 7.8:

```
#* Scene ****************/
# Dominant White Point (D65)
vec3 white = vec3(D65)*100;
white = 0.5 * white + 0.5 * vec3(100,100,100);
# maximum scene luminance
float lmaxA= lmax*100;
# maximum white point
vec3 wmax = white * lmaxA/(lav*100);
```

Listing 7.6: Szene

```
  #*Surround***************/
  # surround luminance
  float ls = 2000;
  # max surround luminance
  float lsMax = 5440;
  # surround white (viewing environment), normed to scene white point
  vec3 ws = yxy2xyz(vec3(ls, 0.3127, 0.3290));
  # max surround white
  vec3 wsMax = yxy2xyz(vec3(lsMax, 0.3127, 0.3290));

```

Listing 7.7: Umfeld

```
  #*Display****************/
  # display luminance
  float ld = 100;
  # max display luminance
  float ldMax = 250;
  # display white
  vec3 wd= yxy2xyz( vec3(ld, 0.302, 0.332));
  # max display white
  vec3 wdMax = yxy2xyz(vec3(ldMax, 0.302, 0.332));

```

Listing 7.8: Monitor

.

Zur Normalisierung müssen die errechneten XYZ-Werte jedoch vor der Farbraumtransformation nach sRGB jeweils durch das Zehnfache von L_{max} dividiert werden.

Zusätzlich zu den Beleuchtungsverhältnissen werden einige weitere Parameter vorgegeben (vgl. Tabelle 7.1 auf der nächsten Seite).

Zwei dieser Parameter sind *targetSize* und *distance*, die zur Berechnung des Betrachtungswinkels benötigt werden (s. Kasten 7.9).

```
'''
/** Diese Methode berechnet den Betrachtungswinkel.
  * @param targetSize   Groesse des Betrachtungsgegenstandes
  * @param distance    Betrachtungsabstand
  *
  * @return angle
  */
  '''
float computeVisualAngle(float targetSize,float distance){

  float angle = 2.0 * atan(targetSize/(2*distance));

  return angle;
}
```

Listing 7.9: Berechnung des Betrachtungswinkels

targetSize gibt dabei die Diagonale des Ausgabemonitors in Zoll an und wird auf 15 gesetzt. *distance* ist der Abstand zum Monitor, ebenfalls in Zoll. Auch dieser Wert wird auf 15 festgelegt. Es handelt sich hierbei jedoch um grobe Annäherungen. Auch ergab

eine Änderung der Bildschirmdiagonalen beispielsweise auf 24 Zoll keine sichtbare Veränderung des Renderings.

Die anderen drei Parameter sind *theta*, *k* und *r*. Sie werden benötigt, um die Sättigungsgrenze der Photorezeptoren V_{max} zu berechnen (s. Kasten 7.10). Die Werte werden auf 66,85 für *theta*, 33,75 für *k* und 0,5 für *r* gesetzt.

```
'''
/** Diese Methode dient zur Berechnung von Vmax
* @param white     White Point
*
* @return vmax
 */'''
vec3 computeVMax(vec3 white){

  float theta = 66.85;
  float k = 33.67;
  float r = 0.5;

  vec3 vmax = vec3(0);

  for(int i =0; i<=2;i++){
    vmax[i]= k * pow((theta + white[i])/theta, -r);
  }
  return vmax;
}
```

Listing 7.10: Methode zur Berechnung von Vmax

Parameter	Wert
theta	66,85
k	33,27
r	0,5
target seize	15
diagonal	15

Tabelle 7.1: Userparameter CIAR

Abbildung 7.2 auf der nächsten Seite zeigt das Ergebnis des CIAR-Shaders, dem das Ergebnis der Matlab-Implementierung in Abbildung 7.3 auf der nächsten Seite gegenüber steht. Letztere stammt von Tania Pouli [58].

Abbildung 7.2: Calibrated Image Appearance Reproduction, GLSL-Shader

Abbildung 7.3: Calibrated Image Appearance Reproduction, Matlab

7.3 Color appearance in high dynamic range imaging

Der von Reinhard et al. vorgestellte Algorithmus [2] führt zunächst ein CAM auf dem Originalbild aus, setzt dann die evtl. veränderte Luminanz zurück und komprimiert diese anschließend durch Tone Mapping. Endprodukt ist somit ein Bild, welches farblich auf die Betrachtungsumgebung angepasst ist, ohne dadurch Details verloren zu haben.

Des weiteren ist dieser Ansatz darauf ausgelegt, jedes beliebige CAM und jeden beliebigen Tone Mapper verwenden zu können. Er wird im Folgenden daher als MixedTMO bezeichnet.

CAM

Als Beispiel für ein CAM wird das CIECAM02-Verfahren implementiert.

Die RGB-Werte werden zunächst in absolute XYZ-Werte umgerechnet. Es folgt die chromatische Adaption, um die Anpassung des Auges an die herrschenden Lichtverhältnisse zu simulieren:

Die XYZ-Werte werden erst in einen sogenannten geschärften RGB-Farbraum umgerechnet. Dann erfolgt eine Skalierung abhängig vom Weißpunkt der Szene und dem Adaptionsgrad D, welche angibt, wie stark das Auge bereits an Lichtverhältnisse der Szene angepasst ist. Die skalierten Werte werden dann wieder in XYZ-Koordinaten zurück transformiert.

Nun erfolgt eine Reihe von Bearbeitungsschritten, mit denen letztendlich die Farberscheinungsattribute berechnet werden sollen:

1 Hilfsparameter: Zunächst werden einige Parameter bestimmt, die zur Berechnung der Farberscheinungsattribute benötigt werden (siehe [2]). Dies sind der Helligkeitsadaptionsfaktor F_L, ein Faktor n für den gesamten Einfluss des Hintergrunds, N_{bb} für den Einfluss der Hintergundhelligkeit und N_{cb} für den Einfluss der Hintergrundfarbe.

2 HPE-Transformation: Die XYZ-Werte werden mit einer geeigneten Matrix in den Hunt-Pointer-Estevéz-Farbraum umgerechnet.

3 Nicht-lineare Kompression: Die HPE-Werte werden über einen von F_L abhängigen Term komprimiert.

4 Farbopponenten: Mit den Hilfsparamtern wird aus den komprimierten HPE-Werten das Farbopponenten-Tripel A, a und b berechnet, welches ähnlich zum CIE Lab Farbraum angelegt ist.

5 Farberscheinungsattribute: Aus diesen Werten können dann die Farberscheinungsattribute berechnet werden (siehe Kapitel 6 auf Seite 36). Sie dienen als Ausgangspunkt für folgende Rücktransformation.

6 Rücktransformation: CIECAM02 ist vollständig invertierbar. Bei der Rücktransformation werden die einzelnen Schritte in der umgekehrten Reihenfolge erneut

durchgeführt, um die Szene schließlich wieder als Bild darstellen zu können. Durch die Anpassung des Weißpunktes und der übrigen Helligkeitsparameter an die Umgebung kann dabei der Farbeindruck auf den Ort angepasst werden, an dem sich der Betrachter befindet.

Tone Mapping

Die lokale Version des fotografischen Tone Mappers von Reinhard et al. wird zur Luminanzkompression verwendet (vgl. Abschnitt 5.2 auf Seite 31). Aus Performancegründen wird in dieser Implementierung des Algorithmus auf die Optimierung durch den Marr-Hildreth-Operator jedoch verzichtet und eine feste Filtergröße von $\sigma = 20$ verwendet.

Implementierung

Die Umsetzung dieses Verfahrens wurde mit Hilfe mehrerer Methoden gelöst (vgl. Kasten 7.11).

```
'''
 Auszug aus der Main-Methode
'''

vec3 xyz = srgb2xyz * srgb;

#luminance values
float yi = xyz.g;


#GamutMapping
vec3 gm = ciecam02(xyz);


#Reset Luminance values
gm.g = yi;

# Transform to srgb (Monitor Color Space)
srgb = inverse(srgb2xyz)*gm;

#ToneMapping
vec3 tonemapped = toneMapping(srgb);


#Final output
srgb = tonemapped;
```

Listing 7.11: Mixed Gamut- and Tone Mapping

Zunächst werden die jeweiligen Pixelwerte in XYZ-Koordinaten umgerechnet und die Luminanz getrennt gespeichert. Das Pixel wird dann der Methode $ciecam02()$ übergeben, welche das Gamut Mapping durchführt.Danach wird der resultierende Y-Wert durch die vorher gespeicherte Luminanz ersetzt, sodass die XYZ-Koordinaten zurück in den sRGB-Raum transformiert werden können. Abschließend wird die Methode $toneMapping()$ auf die Werte ausgeführt, um die Luminanz zu komprimieren.

Zur Farbraumtransformation wird folgende Matrix, bzw. deren Inverse für die Rücktransformation, eingesetzt:

```
mat3 srgb2xyz =   mat3(0.4124,0.2126, 0.0193,
          0.3576, 0.7152, 0.1192,
          0.1805, 0.0722, 0.9505);
```

Listing 7.12: sRGB2XYZ-Matrix

Die Methode $toneMapping()$ ist in Kasten 7.13 dargestellt.

```
'''
  Diese Methode dient als Tone Mapping Funktion.
'''
vec3 toneMapping( vec3 rgb){

  #Luminance
  float lum = dot(vLuminance, rgb);

  #Define key value
  float key = lav;

  #Define Luminance White Point
  float lwhite= 100 * lav;

  #Map Lwhite to key
  float lMapped = (key)/lav * lum;

  #Get surrounding luminance;
  vec4 surround =  blur(10);
  float lSurround = dot(vLuminance, vec3(surround));

  #Calculate displayable luminance
  float lDisplayable = (lMapped * (1.0 + lMapped/(lwhite*lwhite)))
   /(1.0+lSurround);

  #Scale xyz-values according to displayable luminance
  float r = lum/lDisplayable;
  vec3 output=vec3(0);
  output.r = rgb.r/r;
  output.g = rgb.g/r;
  output.b = rgb.b/r;

  return output;
}
```

Listing 7.13: Tone Mapping Methode

Die Luminanz wird zunächst aus den RGB-Werten extrahiert und in der Variable *lum* gespeichert. Dabei gilt:

$$Y = 0.2126R + 0.7152G + 0.0722B. \tag{7.7}$$

Die verwendeten Vorfaktoren sind im Vektor *vLuminance* hinterlegt. Wie im vorherigen Abschnitt beschrieben, wird dieser Luminanzwert dann auf dem Key-Wert der Szene abgebildet und anschließend durch Verwendung einer Sättigungsfunktion komprimiert.

Das Verhältnis von komprimierter Luminanz *lDisplayable* zur Ursprungsluminanz *lum* ergibt den Quotienten, mit dem die RGB-Werte zum Schluss skaliert werden können. Diese werden dann zurück gegeben.

Auch die Methode $ciecam02()$ entspricht dem oben mathematisch beschriebenen Verfahren. Sie ist auf der dieser Arbeit beiliegenden CD dokumentiert.

Dabei ist jedoch zu vermerken, dass zur korrekten Durchführung des Gamut Mappings ein Tiefpassfilter eingesetzt werden muss. Dies ist in einem Shader schwierig umzusetzen, da er den Zugriff auf das gesamte Bild erfordert. Er wird mit dem in Kasten 7.14 beschriebenen Filter angenähert, der nur eine begrenzte, vorher definierte Anzahl an Pixeln um das Zielpixel herum betrachtet. Mehrmalige Filterung des Bildes ist so ohne zwischenzeitliche Speicherung der Textur nicht möglich, jedoch für den Algorithmus auch nicht notwendig.

```
'''
  /** Diese Methode berechnet die Gewichte des Gaussfilters
  * @param pos    Position des Pixels
  * @param sigma  Varianz des Filters
  *
  * @return gaussian
  */
  '''
float computeGaussian(vec2 pos, float sigma){
  float gaussian =0;
  gaussian = 1/(2*(22/7.0)*(sigma * sigma)) * exp(1.0)-(pos.x * pos.x
    +     pos.y * pos.y)/(2*sigma *sigma);
  return gaussian;
}
////////////////////////////////////////////////////////////////////////



'''
  /** Diese Methode filtert das Bild mit einem Gaussfilter
  * @param sigma  Varianz des Filters
  *
  * @return gaussian
  */
  '''
vec4 blur(float sigma){
  int kernelsize=3;
  vec4 pixel = texelFetch( uRadiance, ivec2(gl_FragCoord.xy), 0 );
  vec4 blurred = vec4(0.0);
  float normalize =0;
  for (int i = 0; i< kernelsize; i++){
    int posI = - kernelsize / 2 + i;
    for( int j =0; j< kernelsize; j++){
      int posJ = - kernelsize / 2 + j;
      vec4 tmp = texelFetch( uRadiance, ivec2(gl_FragCoord.xy) +
   ivec2(posI,posJ), 0 );
      float gaussian = computeGaussian(vec2(posI,posJ),sigma);
      blurred += tmp* gaussian;
      normalize += gaussian;
```

```
    }
39  }

41  blurred /= normalize;
    vec4 output = vec4(blurred.rgb, 1);
43
    return output;
45  }
```

Listing 7.14: Tiefpass-Methoden

Die Methode *blur()* liest Pixel um die Position des aktuellen Pixels herum aus der Textur. Die Anzahl der Pixel ist abhängig vom Wert der Variable *kernelsize*, welche die Größe des Filterkernels angibt. Dieser Wert wird auf 3 festgesetzt, was einem 3x3-Filter entspricht. Größere Kerne erzeugen eine Überbetonung der Kanten, kleinere vermindern den Einfluss der Umgebung auf das Tone Mapping.

Durch den Aufruf von *computeGaussian()* wird dann für jedes Pixel das entsprechende Gewicht berechnet. Diese Gewichte werden aufaddiert und anschließend normalisiert.

Parameter: Die Varianz des Filters wird beim Aufruf von *blur()* durch die Variable *sigma* übergeben. Ihr Wert wird auf 10 gesetzt.

Daneben gibt es auch bei diesem Verfahren einige Parameter, die durch den Nutzer bestimmt werden können. Eine Übersicht zeigt Tabelle 7.2 auf Seite 58. Dies sind, abgesehen von den schon angesprochenen Variablen *kernelsize* und *sigma*, zwei Parameter, die im Tone Mapping Anwendung finden, sowie fünf Parameter für das Gamut Mapping.

Die Variable *lwhite* speichert die Luminanz des Weißpunktes. Dieser Wert entspricht ungefähr der Maxmimalluminanz und kann so angenähert werden. Eine weitere Annäherung kann für die Variable *key* getroffen werden, indem sie mit L_{av} gleichgesetzt wird. Auf diese Weise können die Werte direkt aus dem Bild abgeleitet werden, sodass eine Anpassung an verschieden beleuchtete Szenen automatisch erfolgt.

Für das Gamut Mapping werden zunächst die beiden Variablen *la* und *yb* für die chromatische Adaption verwendet. *la* ist der angenommene Luminanzwert, auf den das Auge angepasst ist. Er wird auf 20% des Y-Wertes des Weißpunktes gesetzt. *yb* ist die relative Hintergrundhelligkeit, die auf 0,2 gesetzt wird.

Das CAM erfordert außerdem einen Faktor *c*, der den Einfluss der Umgebung auf das Tone Mapping steuert. Er hat den Wert 0,69. Die letzten beiden Parameter sind f und *nc*, die den Adaptionsgrad des Auges an die Umgebungshelligkeit bzw. die Hintergrundfarbe steuern. Beide erhalten den Wert 1 unter der Annahme, dass das Auge vollständig an die Umgebung angepasst ist.

Um die chromatische Adaption zu vervollständigen, erhalten *yb* und *la* nach dem eigentlichen Gamut Mapping zwei neue Werte, welche die Helligkeit des Displays abbilden. Dies sind 0,16 für *la* und wiederum 0,2 für *yb*.

Die Ergebnisse zeigen Abbildung 7.4 auf der nächsten Seite und Abbildung 7.5 auf der nächsten Seite. Das obere ist das Resultat des Shaders, das untere das der entsprechenden Matlab-Implementierung als Vergleich.

Abbildung 7.4: MixedTMO, GLSL-Shader

Abbildung 7.5: MixedTMO, Matlab

Parameter	Wert
kernelsize	3
sigma	20
lwhite	max luminance
key	log-average luminance
la	0,2 * luminance white point
yb	0,2
c	0,69
nc	1
f	1
la(Display)	0,16
yb(Display)	0,2

Tabelle 7.2: Userparameter MixedTMO

7.4 Color Correction for Tone Reproduction

Der mit „Color Correction for Tone Reproduction“ benannte Algorithmus stellt ein Verfahren dar, welches lediglich die durch das Tone Mapping entstehenden Farbeffekte rückgängig machen soll [59]. Der entsprechende Shader besitzt den Namen „ColorCorrectedTMO“.

Tone Mapping

Der verwendete Tone-Mapping-Operator entspricht dem fotografischen Tone-Mapping-Operator von Reinhard et al. Für eine detaillierte Beschreibung siehe Abschnitt 5.2 auf Seite 31 und [67].

Color Correction

Das Besondere dieses Algorithmus ist die Farbkorrektur, die nach der Luminanzkompression auf das Bild angewandt wird. Dabei handelt sich sich um eine Korrektur des Farbtons und der Sättigung, um speziell durch die Kompression hervorgerufenen Effekten entgegen zu wirken.

Entscheidend ist, dass der Grad an Entsättigung aus der Nichtlinearität der Kompressionskurve geschlossen werden kann, unabhängig davon, ob der Tone Mapper global oder lokal arbeitet. Ausgangspunkt sind zwei lineare Bilder, das Original und die komprimierte Version. Ziel der Farbkorrektur ist es nun, das komprimierte Bild so zu bearbeiten, dass die Erscheinung von Sättigung und Farbton der des Originals entspricht und gleichzeitig die reduzierte Luminanz erhalten bleibt.

Beide Bilder werden dafür zunächst normiert und dann über den *XYZ*-Farbraum in den *IPT*-Farbraum transformiert. Es folgt eine Transformation in einen zylindrischen Farbraum ähnlich zu *CIE LCh*, jedoch auf *IPT* basierend. Die drei Parameter *L*, *C* und *h* entsprechen der Luminanz, Buntheit und dem Farbton des Pixels.

Zuerst wird der Parameter h_1 des korrigierten Bildes gleich dem Wert von h_2 des Originals gesetzt.

Der Buntheit-Wert C_1 des komprimierten Bildes wird zunächst hochskaliert. Aus den Buntheit-Werten C_1 und C_2 wird dann die Sättigung des jeweiligen Bildes bestimmt und zueinander ins Verhältnis gesetzt. Die Sättigung *s* des jeweiligen Bildes ergibt sich durch:

$$s_1 = \frac{C_1}{C_1^2 + L_1^2}. \tag{7.8}$$

Mit diesem Faktor wird nun ebenfalls der Buntheit-Wert C_2 des zweiten Bildes skaliert und dieses anschließend wieder in RGB-Koordinaten zurück transformiert:

$$C_{1-skaliert} = \frac{s_2}{s_1} * C_1. \tag{7.9}$$

Der Luminanz-Wert des korrigierten Bildes bleibt unverändert.

Implementierung

In Kasten 7.15 ist ein Auszug aus der Haupt-Methode des ColorCorrectedTMO-Shaders abgebildet.

```
'''
   Auszug aus der Main-Methode des ColorCorrectedTMO-Shaders
   '''

  vec3 xyz = srgb2xyz * sRGB;

  # ToneMapped Image
  vec3 tonemapped = toneMapping(sRGB);

  vec3 tonemappedXYZ = srgb2xyz * (tonemapped *100);

  # ColorCorrected Image
  xyz = colorCorrection(xyz, tonemappedXYZ);

  # Transformation to sRGB
  sRGB = (xyz2srgb * xyz)/100;
```

Listing 7.15: ColorCorrected Tone Mapping

Maßgeblich für die Funktionalität sind die Methoden *toneMapping()* und *colorCorrection()*.

Die erste führt das eigentliche Tone Mapping durch und funktioniert analog zu Reinhards fotografischem Tone Mapper, welcher ohne Abwandlung übernommen wurde. Die Implementierung entspricht dem Programmcode aus Kasten 7.13.

Die komprimierten Daten werden dann der Methode *colorCorrection()* übergeben, die eine Farbkorrektur auf diesem Bild durchführt. Kasten 7.16 zeigt den dazugehörigen Quelltext.

```
'''
 /** Diese Methode dient zur Farbkorrektur.
 * @param original    Referenzbild, unkomprimierte Vorlage
 * @param tonemapped  komprimiertes Bild
 * @return  xyz farbkorrigiertes, komprimiertes Bild
 *
 * Hier wird das durch Tone Mapping komprimierte Bild in den ICh-
   Farbraum transformiert, um die Farberscheinung an das Original
   anzupassen.
 */
 '''
vec3 colorCorrection(vec3 original, vec3 tonemapped){

  #original image
    #Original LMS
    vec3 lmsO = transformXYZtoLMS(original);

    #Original IPT
    vec3 iptO = transformLMStoIPT(lmsO);

    #Original ICh
```

```
20      vec3 ichO = transformIPTtoICh(iptO);

22
    #tonemapped image
24      #ToneMapped LMS
        vec3 lmsT = transformXYZtoLMS(tonemapped);
26
        #ToneMapped IPT
28      vec3 iptT = transformLMStoIPT(lmsT);

30      #ToneMapped ICh
        vec3 ichT = transformIPTtoICh(iptT);
32

34  #Mapping tonemapped Chroma
    float ctMapped = ichT.g * (ichO.r/ichT.r);
36
    #Saturation
38  float s = ichO.g / sqrt( (ichO.g*ichO.g) + (ichO.r*ichO.r));
    float st = ctMapped / sqrt( (ctMapped * ctMapped) + (ichT.r*ichT.r)
     );
40



42
    #Saturation ratio
44  float r = s/st;

46  #final Chroma
    float cFinal = r * ctMapped;
48
    #final Hue
50  float hFinal = ichO.b;

52  #final Lightness
    float iFinal = ichT.r;
54
    #Transformation to XYZ
56  ichO = vec3(iFinal,cFinal,hFinal);


58
    iptO = transformIChtoIPT(ichO);
60
    lmsO=transformIPTtoLMS(iptO);
62
    vec3 xyz = transformLMStoXYZ(lmsO);
64
    return xyz;
66  }
```

Listing 7.16: Farbkorrektur

Die XYZ-Koordinaten des komprimierten Bildes und des Originals werden zuerst in den IPT- und von dort aus in den ICh-Farbraum transformiert.

Der Chrominanzwert *ichT.g* des komprimierten Bildes wird dann abhängig vom Verhältnis der Intensitäten der beiden Bilder skaliert und in der Variable *ctMapped*

gespeichert. Dieser Wert wird dann ein zweites Mal skaliert. Der zweite Skalierungsfaktor *r* ergibt sich, wie oben beschrieben, aus dem Verhältnis der Sättigungswerte beider Bilder.

Das Endresultat *ichO* setzt sich dann zusammen aus der Intensität *ichT.r* des komprimierten Bildes, dem skalierten Chrominanzwert und dem Farbton *ichO.b* des Originals. Diese drei Werte werden zum Schluss wieder in XYZ-Koordinaten transformiert und zurückgegeben.

Parameter: Die selbst eingestellten Werte der verschiedenen Parameter zeigt Tabelle 7.3 in der Übersicht. Sie entsprechen den Werten des fotografischen TMO, die beim vorhergehenden Algorithmus bereits beschrieben wurden.

Parameter	Wert
Kernelsize	3
sigma	10
lwhite	max luminance
key	log-average luminance

Tabelle 7.3: Userparameter ColorCorrectedTMO

Abbildung 7.6 auf der nächsten Seite und Abbildung 7.7 auf der nächsten Seite zeigen des Ergebnis des Shaders bzw. das der Matlab-Implementierung.

Abbildung 7.6: ColorCorrectedTMO, GLSL-Shader

Abbildung 7.7: ColorCorrectedTMO, Matlab

7.5 Filmic Tone Mapping

Der Filmic Tone Mapper oder auch „FilmicTMO“ wurde von John Hable von Naughty Dog vorgestellt [26]. Er ist an die Empfindlichkeitskurve von Kodak-Filmmaterial und den dadurch entstehenden Filmlook angelehnt. Damit soll gewährleistet sein, dass die Highlights ähnlich zum fotografischen Tone Mapping Operator mit einem weichen Verlauf ins Weiße übergehen, aber gleichzeitig die Sättigung in den Tiefen erhalten bleibt.

Empfindlichkeitskurve

Es handelt sich um einen globalen Operator, der eine Reihe von festgelegten Parametern verwendet, welche die einzelnen Abschnitte der Empfindlichkeitskurve bestimmen bzw. beeinflussen sollen.

Dies sind im einzelnen:

Parameter	Bedeutung	Wert
sStrength	shoulder strength	0,22
linStrength	linear strength	0,30
linAngle	linear angle	0,10
toeStrength	toe strength	0,20
toeNumerator	toe numerator	0,01
toeDenominator	toe denominator	0,30
toeAngle	toe angle	toeNumerator/toeDenominator
white	linear white point	11,2

Tabelle 7.4: Parameter der Film-Empfindlichkeitskurve

Für einen Eingangswert *x* ergibt sich dann die finale Farbe *FinalColor* über:

$$F(x) = \frac{x * k + toeStrength * toeNumerator}{x * n + toeStrength * toeDenominator} - toeAngle \tag{7.10}$$

mit

$$k = x * sStrength + linAngle * linStrength; \tag{7.11}$$

$$n = sStrength * x + linStrength. \tag{7.12}$$

Daraus folgt:

$$FinalColor = \frac{F(x)}{F(white)}. \tag{7.13}$$

Es wird davon ausgegangen, dass es sich bei allen möglichen Werten für *x* um lineare Werte handelt. Daher wird vor der Ausgabe zusätzlich eine Gammakorrektur auf einen Gamma-Wert von 2,2 durchgeführt.

Implementierung

Die Komplexität dieses Verfahrens ist vergleichsweise gering, sodass es durch eine einzige Methode umgesetzt werden kann:

```
'''
  Auszug aus dem FilmicTMO-Shader
'''

# film curve parameters
float sStrength = 0.22;
float linStrength = 0.3;
float linAngle = 0.1;
float toeStrength = 0.2;
float toeNumerator = 0.01;
float toeDenominator = 0.3;
float toeAngle = toeNumerator/toeDenominator;
float white = 11.2;

#texture
  vec4 radiance   = texelFetch( uRadiance, ivec2(gl_FragCoord.xy),
 0 );

#tonemapped values
vec4 tonemapped = radiance;
vec4 colLin = ((tonemapped * (tonemapped * sStrength + linAngle*
 linStrength) + toeStrength*toeNumerator)/(tonemapped * (sStrength
 * tonemapped + linStrength) + toeStrength*toeDenominator)) -
 toeAngle;

float colWhite= ((white * (white * sStrength + linAngle*linStrength
 ) + toeStrength*toeNumerator)/(white * (sStrength * white +
 linStrength) + toeStrength*toeDenominator)) - toeAngle;

FragColor = pow( (colLin/colWhite), vec4(1.0/2.2) );
...
```

Listing 7.17: Filmic Tone Mapping

Auch die Ergebnisse dieses Algorithmus sollen hier kurz demonstriert werden. Das Ergebnis des Shaders zeigt Abbildung 7.8 auf der nächsten Seite, das der Matlab-Implementierung ist in Abbildung 7.9 auf der nächsten Seite zu sehen.

Abbildung 7.8: FilmicTMO, GLSL-Shader

Abbildung 7.9: FilmicTMO, Matlab

7.6 iCam06

Die akkurate Vorhersage der Attribute des menschlichen visuellen Systems für eine möglichst große Spannweite von Bildern ist das Ziel des iCam-Algorithmus. Einsatz findet eine überarbeitete Version aus dem Jahr 2006 [37].

Chromatic Adaption

Als Eingabewerte fungieren XYZ-Tristimulus-Werte in absoluten Einheiten (es gilt: Y = Luminanz). Um feine Strukturen im Bild durch die chromatische Adaption nicht zu beeinflussen, wird das Eingangsbild zunächst in ein Base-Layer und ein Detail-Layer aufgeteilt. Die chromatische Adaption wird dann nur beim Base-Layer verwendet und lässt damit die Details intakt.

Zur Erstellung des Base-Layers wird das Original erst mit einem bilateralen Filter gefiltert. Das Resultat wird dann vom Original abgezogen, um auch das Detail-Layer zu erhalten. Um diese Berechnungen durchzuführen, müssen jedoch alle Pixel-Intensitäten in die Log-Domäne transformiert werden.

Daraus ergeben sich folgende Bearbeitungsschritte:

- Transformation von absoluten Intensitätswerten in Log-Werte
- Base-Layer: bilateral gefiltertes Bild, beschleunigt über stückweise lineare Annäherung und Nearest-Neighbor-Interpolation
- Detail-Layer: Gefiltertes Bild wird vom Originalbild abgezogen
- Rücktransformation beider Layer von Log-Werten zu absoluten Werten

Die chromatische Adaptionsfunktion wird aus CIECAM02 übernommen (vgl. Abschnitt 7.3 auf Seite 52 und [20]). Allerdings wird der chromatische Adaptionsfaktor um den Faktor 0,3 erhöht, um der Entsättigung bei HDR-Bildern entgegen zu wirken.

Tone Compression

Die verwendete Luminanz-Kompression kombiniert die Verhaltensweisen von Stäbchen und Zapfen. Die adaptierten Werte werden zunächst nach XYZ zurück transformiert und anschließend in den HPE- bzw. LMS-Raum transformiert. Dort erfolgt eine nichtlineare Kompression, die erneut an CIECAM02 angelegt ist, jedoch um den Exponenten p erweitert wird. Mit diesem lässt sich die Steilheit der Kompressionskurve steuern.

Die Kompressionskurve ist ebenfalls abhängig von der Umgebungshelligkeit Y_W. Diese ergibt sich aus dem tiefpassgefilterten Original mit einer Filterweite, die einem Drittel der Bildgröße entspricht. Somit kann iCam06 als lokaler Operator angesehen werden.

Detail Enhancement

Separat zum Base-Layer wird das Detail-Layer bearbeitet, um dem Stevens-Effekt gerecht zu werden. Dazu werden die Details mit Hilfe einer Exponentialfunktion abgeschwächt:

$$Details_{neu} = Details^{(F_L+0.8)^{0.25}}. \tag{7.14}$$

Der hier verwendete Parameter F_L ergibt sich aus einer Formel zur Berechnung verschiedener luminanzabhängiger Effekte, die auch in CIECAM02 Verwendung findet.

Image Attribute Adjustments

Beide Layer werden nach der Kompression wieder zusammengefügt und in den IPT-Farbraum konvertiert. Hier werden zunächst P und T mit einem ebenfalls von F_L abhängigen Faktor verstärkt, um dem Hunt-Effekt entgegen zu wirken. Die Verstärkung ist dabei für beide Parameter gleich.

Abschließend wird I über eine Exponentialfunktion verstärkt, um den lokalen Kontrast in Abhängigkeit von der Helligkeit des Bildes anzupassen:

$$I_{neu} = I^{\gamma}. \tag{7.15}$$

Der Wertebereich von γ beschränkt sich jedoch auf die drei Werte 1, 1,2 und 1,5 für eine durchschnittliche, dämmerige oder dunkle Umgebung.

Nach der Anpassung von I, P und T werden die Pixel zurück in den RGB-Farbraum konvertiert und ausgelesen.

Implementierung

Innerhalb des iCam06-Shaders wird zunächst die Hauptmethode aufgerufen. Diese ist in Kasten 7.18 zu sehen.

```
'''
  Main-Methode des iCam06-Shaders
  '''
void main(void)
{
  #Input Original Image
  vec4 radiance = texelFetch( uRadiance, ivec2(gl_FragCoord.xy), 0 );

  vec4 luminances = texelFetch(tex, ivec2(0,0),0);
  lmax = luminances.b;

  vec3 srgb = vec3(radiance);

  vec3 xyz = srgb2xyz*(srgb*1000);

  float sigmaS = 0.02 * uResolution.x/2;
  float sigmaI = 1;

  #Convert to Log
  vec3 logImage = log(xyz)/log(10.0);
```

```
21
   #Base Layer
23 vec3 xyzLogBASE= vec3(bilateralBlur(sigmaS,sigmaI));
   vec3 xyzBASE = vec3(pow(10,xyzLogBASE.r),pow(10,xyzLogBASE.g),pow
     (10,xyzLogBASE.b));
25
   #Detail Layer
27 vec3 xyzLogDETAIL = logImage-xyzLogBASE;
   vec3 xyzDETAIL = vec3(pow(10,xyzLogDETAIL.r),pow(10,xyzLogDETAIL.g)
     ,pow(10,xyzLogDETAIL.b));
29
   #LowPass
31 vec3 xyzLOW = srgb2xyz * (vec3(blur(uResolution.x/2.0))*100);
   vec3 xyzSurround = srgb2xyz * (vec3(blur(uResolution.x/3.0))*100);
33
   #Chromatic adaption
35 vec3 xyzBaseAdapt= chromaticAdaption(xyzBASE,xyzLOW);
   vec3 xyzDetailAdapt= xyzDETAIL;
37
   #Tone mapping
39 vec3 xyzCA = tonecompression(xyzBaseAdapt, xyzSurround)*
     detailAdjustment(xyzDetailAdapt, xyzBASE);
41 #Image attribute adjustements
   xyz =iAA(xyzCA, xyzBASE);
43
   #Convert to sRGB
45 srgb = xyz2srgb * (xyz/10000);
47 #Output
   FragColor = vec4(srgb,1.0);
49 }
```

Listing 7.18: Main-Methode des iCam06-Operators

Die eingelesene Textur wird über eine Matrixmultiplikation in XYZ-Koordinaten transferiert und in der Variable *xyz* gespeichert.

Aus den xyz-Werten werden dann die logarithmierten Pixelintensitäten berechnet und in *logImage* gespeichert.

Als nächstes erfolgt die Erstellung des Base-Layers über den bilateralen Filter. Dieser wird über *bilateralBlur()* aufgerufen und erhält die Varianz des örtlichen Filters *sigmaS* und die des Intensitätsfilters *sigmaI* als Variable übergeben. Als geeignete Werte haben sich 2% der halben Bildbreite für *sigmaS* und 1 für *sigmaI* ergeben. Ein niedrigerer Wert für *sigmaI* führt zu starken Halos an Objekträndern.

Der Kasten 7.19 zeigt den Programmcode des bilateralen Filters.

```
1 '''
   /**
3  * Bilateraler Filter
   * @param sigmaS    Filterbreite für den örtlichen Filter
5  * @param sigmaI    Filterbreite für den Intensitätsfilter
   *
7  * @return output    tiefpassgefiltertes Bild
```

```
  */
  '''
vec4 bilateralBlur(float sigmaS, float sigmaI){

  #Filter size
  int kernelsize =5;
  vec4 blurred = vec4(0.0);
  float normalize =0.0;

  #Go through all kernel pixels
  for (int i = 0; i< kernelsize; i++){
    int posI = - kernelsize / 2 + i;
    for( int j =0; j< kernelsize; j++){
      int posJ = - kernelsize / 2 + j;

      #Get pixel
      vec4 tmp = texelFetch( uRadiance, ivec2(gl_FragCoord.xy)+ivec2(
    posI,posJ), 0 );

      #Get xyz-value
      vec3 xyzTmp = srgb2xyz * (vec3(tmp)*100);

      #Get log-xyz
      vec3 logTmp = log(xyzTmp)/log(10.0);
      float intensity = logTmp.y;

      #Get gaussian weight of the pixels' intensity
      float gaussianI = computeIntensityGaussian(intensity,sigmaI);

      #Get spacial gaussian weight
      float gaussianS = computeGaussian(vec2(posI,posJ),sigmaS);
      float weight = gaussianI * gaussianS;

      #Apply weights
      blurred += weight*vec4(logTmp,1);
      normalize += weight;
    }
  }

  #Normalize
  blurred /= normalize;
  vec4 output = vec4(blurred.rgb,1);
  return output;
}
```

Listing 7.19: Bilateraler Filter

Dieser Filter geht über eine geschachtelte for-Schleife alle Pixel in einem bestimmten Bereich um das Zentrum herum durch. Die Größe des Bereichs ergibt sich auch hier über die Variable *kernelsize*, deren Wert auf 5 festgelegt wurde. Für jedes Pixel in diesem 5x5-Block werden nun die XYZ-Koordinaten ermittelt, der Logarithmus gebildet und dessen Y-Wert in *intensity* gespeichert.

Über *computeIntensityGaussian()* wird nun das Gauß'sche Gewicht zu dieser Intensität ermittelt. *computeGaussian()* liefert das Gewicht des örtlichen Filters. Beide Gewichte

werden anschließend mit dem logarithmischen Pixelwerten multipliziert. Ebenso werden beide Gewichte dem Normalisierungsfaktor *normalize* hinzugefügt.

Nachdem der Pixelblock abgearbeitet ist, wird das Resultat über *normalize* normalisiert und zurückgegeben.

Kasten 7.20 zeigt die Berechnung des Intensitätsfilters. Der örtliche Filter wird mit der gleichen Methode berechnet, die auch beim Verfahren „Color appearance in high dynamic range imaging“ in Abschnitt 7.3 auf Seite 52 Verwendung findet.

```
'''
  Berechnung der Gaussgewichte
  '''
float computeIntensityGaussian(float intensity, float sigma){
  float gaussian =0;
  gaussian = exp(-1.0*(intensity * intensity)/(2.0*sigma *sigma));
  return gaussian;
}
```

Listing 7.20: Gaußfilter für die Pixelintensitäten

Die so berechneten Werte werden in *xyzLogBASE* gespeichert. Das Base-Layer *xyzBASE* erhält man durch Exponentierung der Basis 10 mit diesen Werten. Zur Berechnung des Detail-Layers *xyzDETAIL* wird zunächst *xyzLogBASE* von *logImage* abgezogen und die Differenz ebenfalls als Exponent zur Basis 10 eingesetzt.

Für das Tone Mapping wird das Base-Layer der Methode *tonecompression()* übergeben, *detailAdjustment()* komprimiert die Details. Das Produkt der beiden Rückgabewerte ergibt das komprimierte Bild.

Die Kästen 7.21 und 7.22 zeigen auszugsweise die Berechnung der simulierten Stäbchen- bzw. Zapfenreaktion.

```
'''
  Auszug der Tone Mapping Methode
  '''

  #Exponential function for compressing
  vec3 base = (fl * abs(lms))/smap;
  base = vec3(pow(base.r,p),pow(base.g,p),pow(base.b,p));

  float lmsADAPTEDr = sign(lms.r)*400.0*base.r/(27.13 + base.r) +
    0.1;
  float lmsADAPTEDg = sign(lms.g)*400.0*base.g/(27.13 + base.g) +
    0.1;
  float lmsADAPTEDb = sign(lms.b)*400.0*base.b/(27.13 + base.b) +
    0.1;

  lms = vec3(lmsADAPTEDr, lmsADAPTEDg, lmsADAPTEDb);
```

Listing 7.21: Cone Response

Die Berechnung der Zapfenreaktion erfolgt über eine Exponentialfunktion. *lms* stellt das Base-Layer in LMS-Koordinaten dar. *smap* ist die sogenannte „surround map“. Sie ist über das Helligkeitssignal des tiefpassgefilterten Eingangsbildes gegeben. *fl* ist der

oben beschriebene Einflussfaktor aus CIECAM02. Der Kontrast des Bildes wird über den Faktor *p* gesteuert, für den 0,8 als optimaler Wert ermittelt wurde.

```
'''
  Auszug der Tone Mapping Methode
  '''

  float exponent = pow((fls * s/sw),p);

  #Rod response
  float as = 3.05 * bs * (400 * exponent/(27.13 + exponent)) + 0.03;
```

Listing 7.22: Rod Response

Die Reaktion der Stäbchen wird ähnlich berechnet. Als Basis dient hier jedoch *s/sw*. *s* ist der Helligkeitswert des Pixels, *sw* die angenommene Maximalhelligkeit, die auf 20000 festgelegt wird.

Die hier nicht erklärten Faktoren sind Zwischenwerte, welche die Implementierung vereinfachen.

Die Kompression des Detail-Layers funktioniert analog zum oben beschriebenen mathematischen Verfahren. Auf eine genauere Erläuterung wird hier daher verzichtet. Die vollständigen Tone Mapping Methoden befinden sich auf der CD im Anhang zu dieser Arbeit.

Als letzter Schritt des Algorithmus werden die Farberscheinungsattribute über die Methode *iAA()* überarbeitet. Sie erhält als Parameter das komprimierte Bild über die Variable *xyzCA* sowie das Base-Layer als Referenz. Der Kasten 7.23 veranschaulicht die Berechnung.

```
'''
  Anpassung der Image attributes
  '''
vec3 iAA(vec3 xyz, vec3 xyzBase){
  #Transformation to LMS
  vec3 lms= xyz2lms * xyz*100;
  lms.r = pow(abs(lms.r), 0.43);
  lms.g = pow(abs(lms.g), 0.43);
  lms.b = pow(abs(lms.b), 0.43);

  #Transformation to IPT
  vec3 ipt = lms2ipt * lms;

  #Colorfulness adjustment
    float c = sqrt(ipt.g * ipt.g + ipt.b * ipt.b);

    la = 0.2 * xyzBase.g;
  float k = 1.0/(5.0*la + 1.0);
  float kExp = pow(k,4.0);
  fl = 0.2*kExp*(5.0*la) + 0.1*pow((1.0 - kExp),2.0)*pow((5.0*la)
    ,(1.0/3.0));

  ipt.g=ipt.g*(pow(fl+1.0, 0.15) * ((1.29*c*c - 0.27 * c + 0.42)/ (c*
    c - 0.31 * c + 0.42)));
```

```
23      ipt.b=ipt.b*(pow(fl+1.0, 0.15) * ((1.29*c*c - 0.27 * c + 0.42)/ (
        c*c - 0.31 * c + 0.42)));

25      #surround factor: 1 = average, 1.2 = dim, 1.5 = dark
        float y = 1.00;
27      ipt.r = pow(ipt.r,y);

29    '''
        Inverse modell
31    '''
    #Transform to LMS
33    lms = inverse(lms2ipt) * ipt;

35    #inverse nonlinear compression
    lms.r= pow(abs(lms.r), (1.0/0.43));
37  lms.g= pow(abs(lms.g), (1.0/0.43));
    lms.b= pow(abs(lms.b), (1.0/0.43));
39
    #Convert back to xyz
41  xyz = inverse(xyz2lms) * lms;

43  return xyz;
}
```

Listing 7.23: Image Attribute Adjustment

Nachdem das Bild in den IPT-Farbraum transferiert worden ist, wird zunächst die Chrominanz *c* aus dem P- und T-Wert des Pixels ermittelt. P und T entsprechen nach der Farbraumtransformation dem Grün- bzw. Blaukanal des Pixels. Sie werden anschließend abhängig von *c* verstärkt.

Als zweiter Schritt wird der Intensitätswert I, d.h. der Rotkanal, an die Umgebungshelligkeit angepasst. Dies geschieht über eine Exponentialfunktion mit dem Exponenten *y*. Dieser wird in diesem Fall auf 1 gesetzt, da von einem durchschnittlich hellen Hintergrund ausgegangen werden kann.

Die neuen IPT-Werte werden abschließend wieder in den XYZ-Raum zurück transferiert.

Die korrigierten XYZ-Koordinaten können jetzt über eine erneute Matrixmultiplikation in sRGB-Werte umgerechnet und ausgegeben werden.

Parameter: Tabelle 7.5 auf der nächsten Seite zeigt noch einmal die vom Nutzer definierten Parameter des iCam06-Algorithmus in der Übersicht.

Die beiden Variablen *kernelsize* entsprechen den Größen der Pixelblöcke, die vom Tiefpass- und vom bilateralen Filter bearbeitet werden. Der Tiefpassfilter arbeitet auf einem 3x3 Pixel großen Block, der bilaterale Filter auf 5x5 Pixeln.

sigmaS und *sigmaI* sind die beiden Varianzen des bilateralen Filters. Ihre Werte entsprechen 1% der Bildbreite für den räumlichen Filter und 1 für den Intensitätsfilter.

sigma1 und *sigma2* sind die Varianzen für die beiden Tiefpassfilter, die in iCam06 zusätzlich Verwendung finden. Der Wert von *sigma1* entspricht der halben Bildbreite, der von *sigma2* einem Drittel der Bildbreite.

Parameter	Wert
Kernelsize (gaussian)	3
Kernelsize (bilateral)	5
sigmaS	$0,02 * width/2$
sigmaI	1
sigma1	$width/2$
sigma2	$width/3$
sw	20000
p	0,8
y	1

Tabelle 7.5: Userparameter iCam06

sw ist die maximal mögliche Helligkeit. Als optimaler Wert wurde 20000 ermittelt. Die Steilheit der Kompressionskurve wird über den Faktor *p* bestimmt, der auf 0,8 gesetzt wird. Der Surroundfaktor *y* als letztes erhält ebenfalls den Wert 1.

Die Ergebnisse dieses Verfahrens sind in Abbildung 7.10 auf der nächsten Seite und Abbildung 7.11 auf der nächsten Seite zu sehen. Das obere Bild zeigt ein mit dem vorgestellten Shader berechnetes Rendering. Das untere Bild ist das Ergebnis der Matlab-Implementierung von Jiangtao Kuang und Mark Fairchild. Aufgrund der starken Abweichungen zwischen den beiden Implementierungen und der im Vergleich zu den anderen Verfahren „flauen“ Wirkung der Bilder wird iCam06 nicht für die Nutzerstudie in Kapitel 8 auf Seite 89 herangezogen.

Abbildung 7.10: iCam06, GLSL-Shader

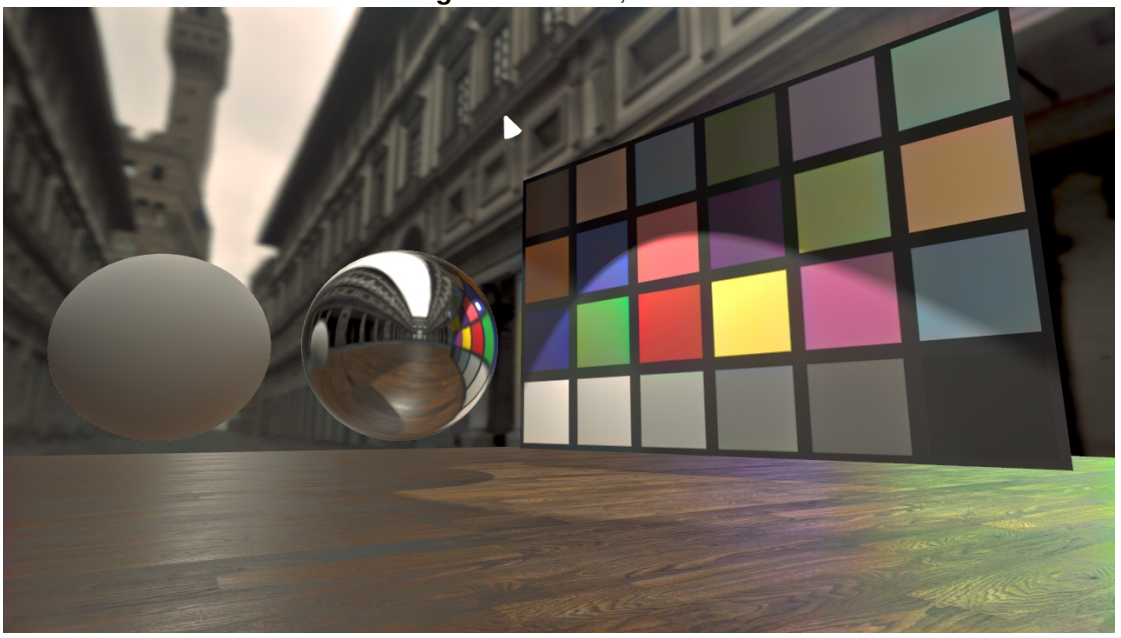


Abbildung 7.11: iCam06, Matlab

7.7 Local Laplacian Filtering

Paris, Hasinoff und Kautz stellen in ihrem Paper [56] bzw. [6] einen Tone-Mapping-Algorithmus vor, der auf einem lokalen Laplace-Filter beruht. In dieser Thesis wird daher die Kurzform „LocalLaplacianTMO“ verwendet.

Der erste Schritt des Verfahrens stellt die Berechnung einer Gauß-Pyramide des Originalbildes dar. Deren Koeffizienten liefern die benötigten Informationen, um dann denjenigen Bereich des Originals einem Remapping zu unterziehen, welcher dem jeweiligen Koeffizienten zugeordnet ist. Aus diesem Zwischenbild wird dann eine Laplace-Pyramide erstellt. Durch einen Schwellenwert können deren Koeffizienten in Details und Kanten unterteilt werden, welche weiter bearbeitet werden können.
Durch die Rekonstruktion der Pyramide erhält man schließlich das Endresultat.

Der Vorteil dieses Verfahrens besteht darin, relativ einfache Formeln zur Reduzierung des Dynamikumfangs verwenden zu können und gleichzeitig durch die Trennung zwischen Details und Kanten eben diese zu erhalten.

Bildpyramiden

Eine Gauß-Pyramide G besteht aus mehreren Bildebenen G_n, die jeweils durch eine Tiefpassfilterung der oberen Ebene entstehen [5]. Ausgangspunkt ist das Originalbild I, welches gleichzeitig die tiefste Ebene der Pyramide darstellt: $G_0 = I$. Durch die Tiefpassfilterung wird jede Dimension des Bildes halbiert, sodass die oberste Ebene der Pyramide nur noch aus wenigen Pixeln besteht.
OpenGL bietet schon eine Implementierung zur Erstellung einer Gauß-Pyramide an. Die MipMaps einer Textur entsprechen hier der jeweiligen Bildebene. Der Zugriff innerhalb des Shaders erfolgt durch:

```
vec4 ebene = texelfetch(textur, position, level);
```

Listing 7.24: MipMap-Abfrage

Die Variable *textur* entspricht dem Originalbild, *position* den Koordinaten des jeweiligen Pixels und *level* dem Index der Pyramidenebene.

Eine Laplace-Pyramide L besteht wie eine Gauß-Pyramide aus mehreren Ebenen L_n, in denen die Frequenzen gespeichert werden, die bei der Erstellung der Gauß-Ebene durch die Tiefpassfilterung verloren gehen. Eine Laplace-Ebene L_n erhält man durch Subtraktion der nächst höheren Gauß-Ebene G_{n+1} von der korrespondierenden Gauß-Ebene G_n:

$$L_n = G_n - upsample(G_{n+1}). \tag{7.16}$$

Dabei ist zu beachten, dass die Ebenen die gleiche Größe haben müssen, weswegen G_{n+1} durch die Methode $upsample()$ vergrößert werden muss. Die höchste Ebene der Laplace-Pyramide entspricht dabei auch der höchsten Ebene der Gauß-Pyramide: $L_{highest} = G_{highest}$. Aus einer Laplace-Pyramide ist das Ursprungsbild wieder rekonstruierbar, indem man die einzelnen Ebenen hochskaliert und aufaddiert.

Remapping

Die Pixel der Laplace-Pyramide werden im nächsten Schritt in Details und Kanten unterteilt, welche dann getrennt verstärkt oder abgeschwächt werden können.

Als Referenz dient das korrespondierende Pixel der Gauß-Pyramide, also der Gauß-Koeffizient der jeweiligen Pyramidenebene an der Stelle $g0 = (x, y)$. Ist die Differenz aus g0 und dem Pixel größer als ein vorher definierter Schwellenwert σ, so handelt es sich um eine Kante. Ist sie kleiner, liegen Details vor. Paris et. al. schlagen als Schwellenwert $\sigma = \log(2, 5)$ vor.

Abhängig von diesem Schwellenwert wird auf alle Pixel der zu $g0$ gehörigen Region des Originalbildes eine Remapping-Funktion $r(x)$ angewendet.

Das Remapping bei Details erfolgt durch

$$r_d(p(x,y)) = g0 + sign(p(x,y) - g0) * \sigma * details(\frac{\mid p(x,y) - g0 \mid}{\sigma}). \tag{7.17}$$

Bei Kanten ist $r(p(x, y)) = r_e(p(x, y))$ mit:

$$r_e(x) = g0 + sign(p(x,y) - g0) * (edges(\mid p(x,y) - g0 \mid -\sigma) + \sigma). \tag{7.18}$$

$p(x, y)$ entspricht in beiden Fällen dem Luminanzwert an der Stelle (x,y) im Originalbild.

Die Methoden details() und edges() sind wie folgt definiert:

details(): Die Verstärkung der Details erfolgt durch eine Exponentialfunktion der Form

$$details(x) = \tau * \alpha^x + (1 - \tau) * x. \tag{7.19}$$

Hier entspricht α einem vom Nutzer definierten Verstärkungsfaktor. Über τ kann der Einfluss von Rauschen auf das Ergebnis reduziert werden. Dieser Parameter wird über ein beliebiges Smooth-Step-Verfahren ermittelt. Implementiert wurde das Verfahren von Ken Perlin [18]. Es gilt:

$$\tau = 0, \text{ falls } x < edge1 \tag{7.20}$$

$$\tau = 1, \text{ falls } x > edge2 \tag{7.21}$$

$$\tau = (x * (x * 6 - 15) + 10) * x^3 \text{ sonst.} \tag{7.22}$$

edges(): Zur Bearbeitung der Kanten wird eine einfache Multiplikation mit einem durch den Nutzer definierten Faktor β durchgeführt:

$$edges(x) = \beta * x. \tag{7.23}$$

β sollte beim Tone Mapping kleiner als 1 sein, da die Kanten zwar teilweise erhalten, aber nicht verstärkt werden sollen.

Die bisherigen Schritte dienen in erster Linie zur Differenzierung und Bearbeitung von Details und Kanten bzw. zur angepassten Erstellung der Bildpyramiden. In dieser Form können sie laut Paris et al. [56] für diverse Bildverarbeitungsprozesse angewandt werden, ohne dabei Kanten negativ zu beeinflussen. Das eigentliche Tone Mapping erfolgt anschließend.

Tone Mapping

Die Autoren des Papers [56] sehen ein relativ einfaches Tone-Mapping-Verfahren vor, welches den Dynamikumfang auf einen vorher definierten Wert festlegt.

Dazu wird zunächst der tatsächlich im Bild vorhandene Dynamikumfang ermittelt, indem das Luminanzminimum vom Maximum abgezogen wird. Verwendet wird jedoch nicht das absolute Maximum und Minimum, sondern 99,5 Prozent bzw. 0,5 Prozent von L_{max}.

Anschließend wird ein Skalierungsfaktor ermittelt, indem der erwünschte Kontrast durch diese Differenz dividiert wird:

$$\gamma = \frac{contrast}{L_{max} - L_{min}}. \tag{7.24}$$

Mit diesem Wert werden die Pixel des durch das Remapping entstandenen Zwischenbildes multipliziert. Anschließend kann die Laplace-Pyramide erstellt und wieder rekonstruiert werden.

Implementierung

Die Implementierung dieses Verfahrens unterscheidet sich in einigen Punkten deutlich von der mathematisch beschriebenen Vorgehensweise. Dies ist der parallelen Rechenarchitektur geschuldet.

Kasten 7.25 zeigt die Hauptmethode des Shaders.

```
'''
  Main-Methode des Shaders
  '''
void main(void)
{
  #Get input image
    vec4 radiance    = texelFetch( uRadiance, ivec2(gl_FragCoord.xy),
   0 );
    vec3 tonemapped = vec3(radiance.r, radiance.g, radiance.b);


    #Get luminances
    vec4 luminances = texelFetch(tex, ivec2(0,0),0);
  lav = luminances.r*100;
  lmin = luminances.g*100;
  lmax = luminances.b*100;


  level1 = upscale(0)-upscale(1);
  level2 = upscale(1)-upscale(2);
  level3 = upscale(2)-upscale(3);
  level4 = upscale(3)-upscale(4);
  level5 = upscale(4)-upscale(5);

  #Apply tone mapping
  tonemapped = tonemapping(tonemapped);

  #Apply gamma correction
```

```
  tonemapped = pow(tonemapped, vec3(1.0/2.2));

  #tonemapped = vec3(textureLod( uRadiance, vTexCoord, 0.5f));



  #return
  FragColor = vec4(tonemapped,1.0);
}
```

Listing 7.25: Hauptmethode des LocalLaplacianFiltering-Shaders

Zunächst werden das Bild sowie die Luminanztextur eingelesen und in den Variablen *tonemapped* bzw. *luminances* gespeichert. Bei den Variablen *level1* bis einschließlich *level5* handelt es sich um die Laplace-Level. Diese werden durch Subtraktion des korrespondierenden MipMap-Levels von dem nächst niedrigeren berechnet, wobei die Methode *upscale(level)* das MipMap-Level mit dem Index *level* auf die Größe des Originalbildes skaliert. Bei der Implementierung wird das Verfahren auf die Nutzung der untersten fünf MipMap-Level beschränkt.

Dieses Upscaling ist notwendig, um einheitliche Pixelkoordinaten bei den einzelnen Levels verwenden zu können. Es wird über einen 3x3-Gaußfilter umgesetzt, wie Kasten 7.26 veranschaulicht:

```
'''/**
 * Diese Methode dient dazu, ein angegebenes Mipmap-Level auf die
   Groesse des Originalbildes zu skalieren.
 * @param   level   der Index der Mipmap-Stufe
 * @return  outPixel  das hochskalierte Ergebnis
 */ '''
vec3 upscaling(int level){
  vec2 coords = gl_FragCoord.xy/(pow(2f,float(level)));
  vec4 center = texelFetch(uRadiance,ivec2(coords),level);
  vec4 top = texelFetch(uRadiance,ivec2(coords + vec2(0,1)),level);
  vec4 topLeft =texelFetch(uRadiance,ivec2(coords + vec2(-1,1)),level
   );
  vec4 topRight =texelFetch(uRadiance,ivec2(coords + vec2(1,1)),level
   );
  vec4 bottom = texelFetch(uRadiance,ivec2(coords + vec2(0,-1)),level
   );
  vec4 bottomLeft =texelFetch(uRadiance,ivec2(coords + vec2(-1,-1)),
   level);
  vec4 bottomRight =texelFetch(uRadiance,ivec2(coords + vec2(1,-1)),
   level);
  vec4 left = texelFetch(uRadiance,ivec2(coords + vec2(-1,0)),level);
  vec4 right = texelFetch(uRadiance,ivec2(coords + vec2(1,0)),level);

  vec4 pixel = (4*center + 2*top + 2*bottom + 2*left + 2*right+
   topLeft + topRight + bottomLeft + bottomRight)/16;

  vec3 outPixel= vec3(pixel);

  return outPixel;
  }
```

Listing 7.26: Skalierungsmethode

Der Vektor *coords* erhält die dem MipMap-Level angepassten Koordinaten. Das entsprechende Pixel sowie alle acht angrenzenden Pixel werden abgefragt und gewichtet aufaddiert.

Um auf die Mipmap-Level zugreifen zu können, wird die Renderpass-Liste für die Luminanzbildberechnung um zwei weitere Passes erweitert. Diese haben keinen Einfluss auf das Bild selbst, erzeugen aber über den passenden Callback die Mipmap-Stufen. Der Callback ist in Kasten 7.27 dargestellt.

```
genMipmapCallback = new RenderCallback()
      {
          @Override
          public void onRender( Mat4 viewMatrix, Mat4 projMatrix,
    ArrayList<GLMesh> meshes, ArrayList<Material> materials, ArrayList
    <Mat4> transforms, GLTexture[] userdata, int resourceIndex)
          {

              GL30.glBindFramebuffer(GL30.GL_FRAMEBUFFER,
    framebuffer2ID);

              for( GLMesh mesh : meshes )
                      mesh.draw();

              GL30.glBindFramebuffer(GL30.GL_FRAMEBUFFER,0);
              GL13.glActiveTexture(GL13.GL_TEXTURE0);
              GL30.glGenerateMipmap(GL_TEXTURE_2D);
              glBindTexture(GL_TEXTURE_2D, mipmapTextureID);
           }
};
```

Listing 7.27: Callback zur Erzeugung der Mipmaps

Über *GL30.glGenerateMipmap()* werden die Mipmaps erzeugt. Diese können für die Originaltextur nicht direkt aktiviert werden, weswegen der Umweg über das zweite FBO gemacht werden muss.

Über den Aufruf der Methode *tonemapping()* wird als nächstes die Filterung einschließlich Luminanzkompression durchgeführt. Hierzu wird pro Pixel das korrespondierende Pixel jeder Laplace-Ebene der Filtermethode *filtering()* übergeben. Deren Rückgabewerte werden aufaddiert, bis jede Laplace-Ebene bearbeitet wurde. Die Summe wird anschließend der Methode *compress()* übergeben, welche die Luminanzkompression durchführt. Den Ablauf innerhalb der *tonemapping()*-Methode zeigt Kasten 7.28 auf der nächsten Seite.

```
'''
/**
* Diese Methode dient der Laplace-Filterung.
* @param   original   das unkomprimierte Eingangsbild
*
* @return  output  das gefilterte und komprimierte Bild
*
*/
'''
 vec3 tonemapping(vec3 original){
 vec3 output = vec3(1.0);

 #Image pixel
 vec3 inPixel = original;

 #Get log luminance of from original values
 float luminance = dot(inPixel,vLuminance);
 logLuminance = log(luminance);
 vec3 tmpPixel = upscale(5);
 float tmpLum = dot(tmpPixel, vLuminance);

 float laplaceLum = log(tmpLum);

 float tmpLogLum = 0.0;
 vec2 coords = vec2(0);

 #highest level
 for(int i =4; i>=0; i--)
 {
   vec3 tmpPixel = vec3(0);
   switch(i){
   case 4: tmpPixel = level5;
        break;
   case 3: tmpPixel =level4;
        break;
   case 2: tmpPixel =level3;
        break;
   case 1: tmpPixel =level2;
        break;
   case 0: tmpPixel =level1;
        break;
   default: tmpPixel =level1;
        break;
   }

   tmpLogLum = dot(tmpPixel,vLuminance);
   laplaceLum += filtering(i, tmpLogLum);
 }

 #simple tone mapping / postprocessing
 laplaceLum = exp(laplaceLum);
 float tonemappedLum = compress(laplaceLum);

```

```
55    #degree of compression
      float scale = luminance/tonemappedLum;
57
      #scale pixel according to compression
59    output = inPixel/scale;

61    return output;
}
```

Listing 7.28: Tone Mapping Methode

Die *filtering()*-Methode entspricht dem unter „Remapping“ beschriebenen Verfahren. Dieses wurde unverändert übernommen. Dies gilt auch für die Methode *compress()*, deren Funktionsweise unter „Tone Mapping“ beschrieben wird.

Parameter: Implementiert wurden hier die Parameter, die in Tabelle 7.6 aufgelistet sind. Als Zielkontrast *contrast* hat sich ein Wert von 5 als optimal erwiesen. Der

Parameter	Wert
contrast	5
sigma	ln(2,5)
edge0	0,01
edge1	0,02
alpha	1,0
beta	0,0

Tabelle 7.6: Userparameter LocalLaplacian

Threshold zur Differenzierung von Kanten und Details wurde auf den von den Autoren empfohlenen Wert von ln(2,5) gesetzt. *edge0* mit 0,01 und *edge1* mit 0,02 sind die beiden Grenzwerte der Smoothstep-Funktion. *alpha* ist der Verstärkungsfaktor der Details, für den ein optimaler Wert von 1 ermittelt wurde. Der Faktor für die Kanten *beta* ist auf 0 gesetzt. Diese werden so nicht verändert.

Das Bild in Abbildung 7.12 auf der nächsten Seite zeigt das Ergebnis des Shaders. Auffällig sind die starken Artefakte, welche vom Upsampling herrühren. Eine nähere Untersuchung findet sich in Abschnitt 8.2 auf Seite 95. Als Vergleich dazu zeigt das untere Bild 7.13 das Ergebnis der Matlab-Implementierung. Diese Implementierung stammt von den Autoren dieses Verfahrens (vgl. [56] bzw. für den Matlab-Code [57]). Sie wurde unverändert auf ein HDR-Screenshot aus dem Renderer angewandt.

Die deutlich sichtbaren Artefakte der Shader-Implementierung sind der Skalierungsmethode geschuldet. Aufgrund der Artefakte und sichtbarer Fehlfarben in dunklen Bildbereichen wird auch dieses Verfahren von der Nutzerstudie ausgeschlossen.

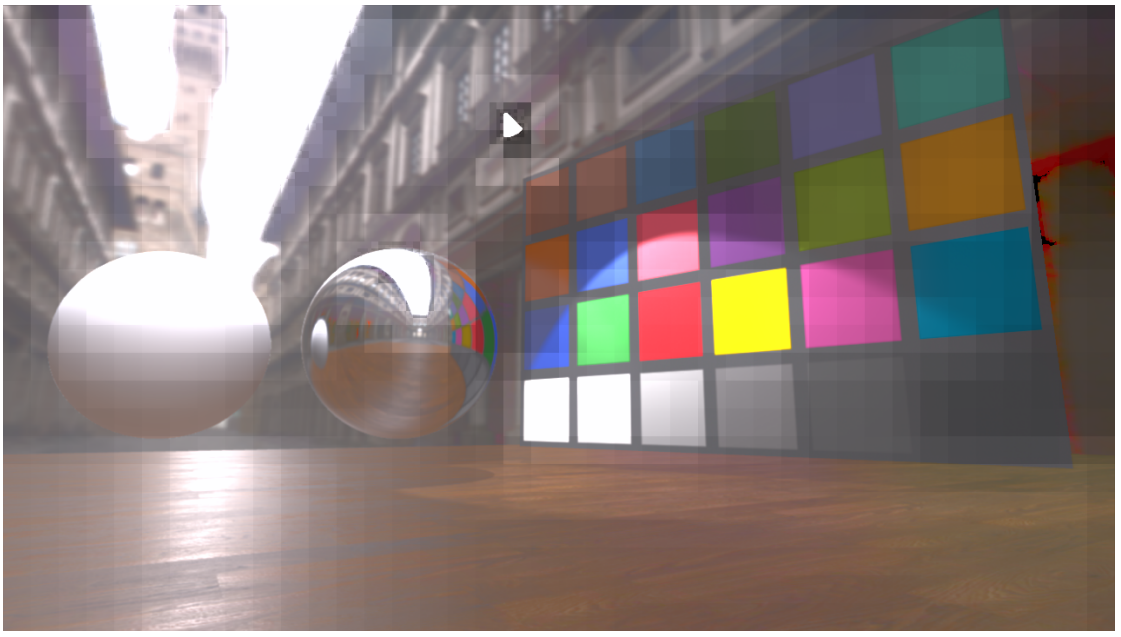


Abbildung 7.12: LocalLaplacianFilterTMO, GLSL-Shader

Abbildung 7.13: LocalLaplacianFilterTMO, Matlab

7.8 Eigener Operator

Grundideen

Als Basis zur Erstellung des eigenen TM-Algorithmus dienen die folgenden Ansätze:

- Der Dynamikumfang soll durch eine Sättigungsfunktion komprimiert werden, die der Funktionsweise der Photorezeptoren entspricht. Effekte wie Blendung o.ä. sollen dabei unberücksichtigt bleiben.
- Zur Korrektur der Farberscheinungseffekte sollen die betroffenen Attribute nach dem Tone Mapping auf die Werte des unkomprimierten Bildes zurückgesetzt werden.

Dieser Tone Mapper erhält den Namen „MyTMO“.

Tone Mapping

Als Kompressionsfunktion wird der einfache TMO von Reinhard et al. verwendet:

$$L_{compressed}(x,y) = white * \frac{luminance(x,y)^n}{1 + y_{Surround}^n}. \tag{7.25}$$

Hier entspricht $luminance(x,y)$ der Ursprungsluminanz des Pixels k an der Stelle (x,y) und $white$ der nutzerdefinierten Helligkeit des Weißpunktes. Im vorliegenden Fall ist $white = 1,2$. Dieser Wert stellt eine angenehme Helligkeitswirkung dar. Ab einem Wert von 1,5 ist an einigen sehr hellen Stellen im Bild keine Zeichnung mehr zu erkennen.

Der Parameter $y_{Surround}$ stellt die Umgebungshelligkeit des Pixels dar, die über einen Gaußfilter mit einer Varianz von $\sigma = 10$ ermittelt wird. Mit dem Exponenten n als Sensitivitätsregler kann die Steilheit der Kompressionskurve und daraus resultierend der Kontrast des Bildes angepasst werden. Als optimaler Wert für n wurde durch empirische Untersuchungen 0,75 ermittelt. Bei größeren Werten wirkt das Bild zu kontrastreich und es sind an einigen dunklen Stellen keine Details mehr erkennbar. Bei kleineren Werten wirkt das Bild dagegen sehr „flach“.

Gamut Mapping

Analog zum Algorithmus in Abschnitt 7.4 auf Seite 59 erfolgt das Gamut Mapping durch Postprocessing nach der Luminanz-Kompression, um dadurch entstandende Artefakte rückgängig zu machen.

Zunächst werden sowohl das Originalbild als auch das korrigierte Bild vom RGB-Farbraum in den HSL-Farbraum transformiert. Der Farbton und die Sättigung werden nun vom Original unverändert übernommen, während der Luminanz-Wert des korrigierten Bildes für das Endergebnis verwendet wird. Aus diesem Tripel werden dann wieder RGB-Werte berechnet.

Implementierung

Beim zweiten Durchlauf dieses Shaders wird zunächst die Methode „tm(radiance, luminance, lighting)“ aufgerufen, welche das Tone Mapping analog zum oben beschriebenen Verfahren durchführt:

```
  '''
 /**
  * Diese Methode dient dem Tone Mapping.
  * @param   radiance   das unkomprimierte Eingangsbild
  * @param   luminance Luminanz des Eingangsbildes
  * @param   lighting  int-Wert, der zwischen photopischen,
    mesotopischen und skotopischen Lichtverhaeltnissen differenziert.
  * @return  photopic  das komprimierte Bild
  *
  * Hier wird die Luminanz lokal über eine Sättigungsfunktion
    komprimiert und
  * die RGB-Werte entsprechend angepasst.
  */
  '''
vec3 tm(vec3 radiance, float luminance, int lighting){

  #Original
  vec3 rgb= radiance;

  #Max white value
  float white = 0.2126 + 0.7152 + 0.0722;

  #Surround value: gaussian blur with blur(sigma)
  vec4 surround = blur(20);


  #Surround luminance
  float ySurround = 0.2126*surround.r + 0.7152*surround.g + 0.0722*
    surround.b;

  float y = sqrt(ySurround * luminance);

  #sensitivity controll exponent
  float n = 0.75;

  #Compressed luminance
  float yMapped =0;
  yMapped = (white) * pow(luminance,n) / (1 + pow(y,n));

  #Compression ratio
  float ratio = yMapped/luminance;

  #Output
  vec3 tonemapped = vec3(1.0);
  tonemapped = vec3(rgb.r * ratio, rgb.g*ratio, rgb.b*ratio);

  return tonemapped;
}
```

Listing 7.29: Tone Mapping

Für den Umgebungsdurchschnitt wird ein Gaußfilter auf das Bild gelegt und das Ergebnis in der Variable $surround$ gespeichert. $ySurround$ bekommt daraufhin die zugehörige Helligkeit zugewiesen. Der Gaußfilter arbeitet dabei ebenfalls auf einem 3x3-Pixel großen Kernel.

Für die Farbkorrektur werden die HDR-Textur sowie das komprimierte Bild jeweils in den HSL-Farbraum konvertiert. Nun wird die unkomprimierte Luminanz durch die Luminanz lum des getonemappten Bildes ersetzt und das Bild anschließend wieder in den linearen sRGB-Farbraum konvertiert (vgl. Kasten 7.30).

```
'''
/**
 * Diese Methode dient dem Gamut Mapping.
 * @param image   komprimiertes Bild
 * @param reference Referenzbild, unkomprimierte Vorlage
 * @return  gamutmapped farbkorrigiertes, komprimiertes Bild
 *
 * Hier wird das durch Tone Mapping komprimierte Bild in Sättigung
   und Farbton
 * der unkomprimierten Referenz angepasst.
 */'''
vec3 gm(vec3 image, vec3 reference){
  vec3 gamutmapped = vec3(0);

  #RGB to HSV conversion
  float cMax = max(max(reference.r,reference.g),reference.b);
  float cMin = min(min(reference.r,reference.g),reference.b);
  float delta = cMax-cMin;

  #Luminance
  float lum = 0.2126*image.r + 0.7152*image.g + 0.0722*image.b;

  #Saturation
  float saturation = 0;
  if(delta != 0){
    saturation = delta/cMax;
  }

  #Hue
  float hue = 0;
  if (cMax == reference.r){
    hue= 60 * mod(((reference.g - reference.b)/delta),6);
  }else if (cMax == reference.g){
      hue= 60 *(((reference.b - reference.r)/delta)+2);
    }else {
      hue= 60 *(((reference.r - reference.g)/delta)+4);
  }


  #Inverse transformation
  float c = lum * saturation;
  float x = c *(1-abs(mod((hue/60),2)-1));
  float m = lum - c*0.5;

```

```
  vec3 rgb = vec3(0);

  if((60>hue) && (hue>=0))
    { rgb = vec3(c, x, 0);}
  else if((120>hue) && (hue>=60))
      { rgb = vec3(x,c,0);}
    else if((180>hue) && (hue>=120))
        { rgb = vec3(0,c,x);}
      else if((240>hue) && (hue>=180))
          { rgb = vec3(0,x,c);}
        else if((300>hue) && (hue>=240))
            { rgb = vec3(x,0,c);}
          else
            { rgb = vec3(c,0,x);}

  #Output
  gamutmapped = vec3((rgb.r + m), (rgb.g + m), (rgb.b + m));

  return gamutmapped;
}
```

Listing 7.30: Gamut Mapping

Das Bild kann dann ausgegeben werden.

Parameter: Tabelle 7.7 zeigt die verwendeten Parameter in der Übersicht.

Parameter	Wert
white	1
n	0,75
sigma	20
Kernelsize	3

Tabelle 7.7: Userparameter MyTM

Das Ergebnis ist in Abbildung 7.14 auf der nächsten Seite zu sehen. Als direkten Vergleich stellt das Bild 7.15 darunter das Ergebnis der Matlab-Implementierung dar.

Abbildung 7.14: Eigener Operator, GLSL-Shader

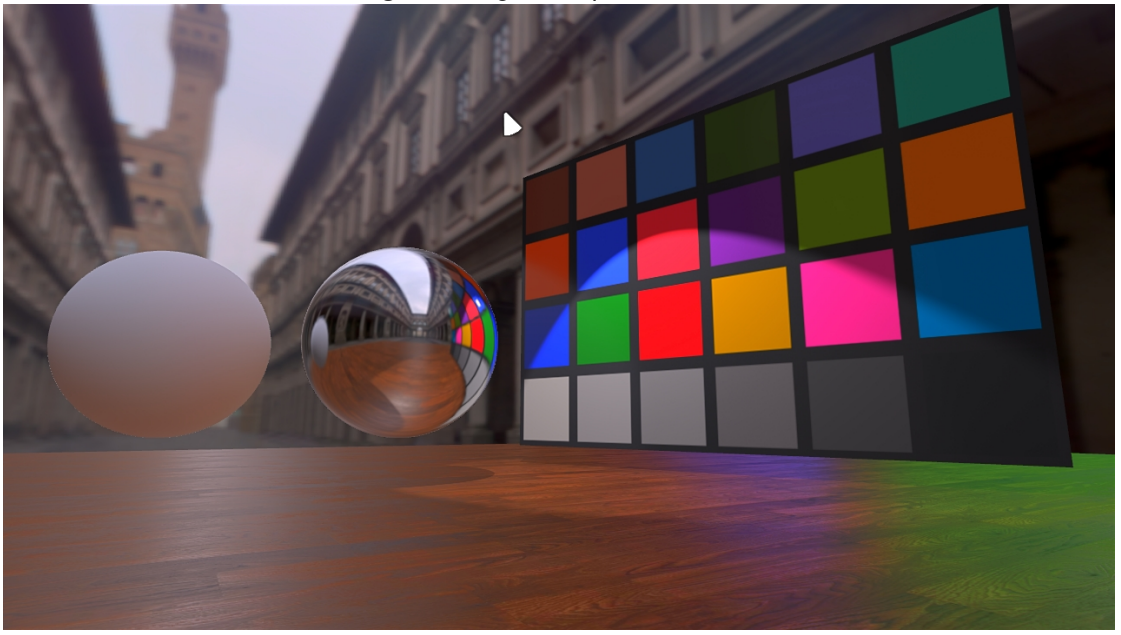


Abbildung 7.15: Eigener Operator, Matlab

8 Evaluierung

Für die Bewertung der vorgestellten Verfahren spielen vor allem zwei Punkte eine entscheidende Rolle. Das ist zum einen die Performance, die den Ansprüchen an Echtzeitanwendungen genügen muss, und zum anderen die Qualität der Ergebnisse als Maß für die Ähnlichkeit der Renderings mit realen Bildern.

8.1 Performance

Die Performance stellt ein Kriterium dar, mit dem sich verschiedene Algorithmen objektiv vergleichen lassen.

Maßgeblich für die Performance von Programmen oder einzelnen Funktionen ist deren Laufzeit, d.h. diejenige Zeit, die benötigt wird, um eine Aufgabe zu lösen. Allgemein gilt: Je schneller ein Programm arbeitet, desto besser.

Für Echtzeit-Anwendungen kommt hinzu, dass die gesamte Laufzeit 16 ms nicht überschreiten darf. Dies entspricht einer Bildwiederholrate von 60 Bildern pro Sekunde, die für eine ruckelfreie Darstellung notwendig ist.

8.1.1 Apparatus

Für die Laufzeitmessung wurde folgende Hardware verwendet:

Komponente	Daten
CPU	AMD FX(tm)-8320 Octacore 3,5 GHz
Arbeitsspeicher	16 GB
Grafikkarte	GeForce GTX 650 128-bit GDDR5 1024MB
Betriebssystem	Ubuntu 14.04 64 Bit

Tabelle 8.1: Plattform Laufzeitmessung

8.1.2 Zeitmessung

Für die Messung der Laufzeit wird ein Verfahren benutzt, welches in Kasten 8.1 veranschaulicht wird.

```
double lastTime = System.currentTimeMillis();
int nbFrames = 0;

double renderTime = 0;

//Main loop
while(render){
  ...
  for(everyFrame){

    ...
    // Measure speed
    renderTime = System.currentTimeMillis();
    rendering();
    renderTime = (System.currentTimeMillis()-renderTime);
  }
  double currentTime = System.currentTimeMillis();
  // If last print was more than 1 sec ago:
  if ( currentTime - lastTime >= 1000.0 )
    {
      // print and reset timer
      System.out.println("ms/frame :" + renderTime);
      nbFrames = 0;
      lastTime += 1000.0;
    }
}
```

Listing 8.1: Zeitmessung

Innerhalb des Programms wird vor der Renderschleife eine Double-Variable mit dem Namen *renderTime* initialisiert. In der Render-Schleife wird dann vor dem eigentlichen Renderer-Aufruf über *System.currentTimeMillis()* die aktuelle Systemzeit abgefragt und in *renderTime* gespeichert. Nach dem Aufruf des Renderers wird erneut die Systemzeit abgefragt und der alte Wert vom neuen abgezogen. Die Differenz ist die Rechenzeit, die der Renderer benötigt.

Außerdem wird auch am Anfang des Programms die Systemzeit erfasst und in der Variable *lastTime* gespeichert. Durch eine erneute Zeitabfrage und Differenzbildung nach jedem Schleifendurchlauf wird überprüft, ob die vergangene Zeit größer als 1 Sekunde ist. Falls das zutrifft, wird die ermittelte Rechenzeit in der Konsole ausgegeben und *lastTime* um 1000 Millisekunden erhöht.

Mit diesem Verfahren kann nicht nur die Performance des Renderers erfasst werden. Auch diejenige einer beliebigen anderen Funktion kann überprüft werden, indem jeweils vorher und nachher die Zeit gemessen wird.

Allerdings ist zu beachten, dass es sich bei den ermittelten Werten um absolute Zeitangaben handelt. Laufen im Hintergrund weitere Prozesse ab, etwa durch andere Programme, können diese Einfluss auf die Rechenleistung und damit auf die Laufzeitmessung haben. Für die Messung wurde daher darauf geachtet, die Hintergrundprozesse möglichst zu minimieren und die Auslastung des Rechners konstant zu halten.

Laufzeiten der Shader

Die Laufzeit der einzelnen Shader wurde wie oben beschrieben gemessen und gegenübergestellt. Die Ergebnisse fasst Tabelle 8.2 zusammen.

Shader	ø Laufzeit
StandardTMO	1,3898 ms
ColorCorrectedTMO	66,4915 ms
MixedTMO	66,5763 ms
iCam06	83,1695 ms
CIAR	66,4237 ms
FilmicTMO	3,4068 ms
MyTMO	3,2542 ms
LocalLaplacianTMO	62 ms

Tabelle 8.2: Laufzeiten der einzelnen Shader

Auf den ersten Blick fällt auf, dass die Shader sich anhand der Laufzeit in zwei Gruppen aufteilen lassen. Zum einen erzielt die Gruppe aus FilmicTMO und MyTMO eine relativ hohe Performance, sodass die Laufzeiten der Shader sich in der Nähe zum StandardTMO als Referenz bewegen. Hier beträgt der Unterschied lediglich 2 ms.

Die anderen Shader bilden die zweite Gruppe, deren Laufzeiten deutlich größer sind. Genauer gesagt sind die Laufzeiten sogar so groß, dass keiner dieser Shader der Echtzeitanforderung von 60 Hz bzw. 16 ms pro Frame gerecht wird.

Bei der Suche nach der Ursache dieses Unterschiedes hilft ein Blick auf die Funktionsweise der einzelnen Shader. Der StandardTMO sowie der FilmicTMO sind sehr simple Algorithmen. Das jeweilige Verfahren zur Luminanzkompression ist vollkommen unabhängig von der Umgebung des jeweiligen Pixels und kommt ohne globale Bildparameter aus. MyTMO arbeitet lokal und ist somit etwas komplexer. Jedoch benötigt dieser Algorithmus zur Kompression lediglich Informationen über die direkte Nachbarschaft des jeweiligen Pixels. Diese sind auf der Grafikkarte schnell zu berechnen.

Alle übrigen Algorithmen beruhen zumindest teilweise auf Parametern, die sich nur durch Verarbeitung des gesamten Bildes berechnen lassen. Diese müssen über die CPU ermittelt werden, wodurch der Vorteil der Parallelisierung verloren geht. Daraus ist zu folgern, dass der Verlust an Performance durch diejenigen Methoden hervorgerufen wird, die auf der CPU stattfinden.

Überprüfung der Luminanzberechnung

Eine dieser Methoden ist *calculateLuminances()*, mit der die Maximal-, Minimal- und Durchschnittsluminanz berechnet wird. Berücksichtigt man bei der Laufzeitmessung

nur diese Funktion, erhält man eine durchschnittliche Zeit von 0,4068 ms pro Frame. Dadurch alleine ist der große Laufzeitunterschied folglich nicht zu erklären.

Überprüfung des Pixeltransfers

Eine weitere dieser Methoden stellt die Methode zum Auslesen der Textur dar, welche verwendet wird, um L_{max}, L_{min} und L_{av} zu bestimmen:

```
glReadPixels(int x, int y, int width, int height, int format, int
    type, Buffer data);
```

Listing 8.2: Methode zum Auslesen der Textur

Hier stehen x und y für die Position des ersten Pixels in der Textur sowie $width$ und $height$ für die Breite bzw. Höhe in Pixeln des auszulesenden Ausschnittes. Über $format$ wird das Pixelformat definiert (z.B. RGBA, RGB oder BGR), $type$ legt den Datentypen fest und $data$ ist ein Buffer-Objekt, in dem die Werte gespeichert werden.

Die Laufzeit dieser Funktion ist sehr hoch. Als Durchschnitt wurde ein Wert von ca. 77 ms ermittelt. Dieser Wert ist abhängig von mehreren Faktoren. Eine wichtige Rolle spielt zum einen die Größe der auszulesenden Region, also die Anzahl der Pixel, die übertragen werden müssen. Tabelle 8.3 verdeutlicht diesen Zusammenhang.

Größe	ø Laufzeit
1 Bildgröße	139,2034 ms
$\frac{1}{4}$ Bildgröße	78,6271 ms
$\frac{1}{16}$ Bildgröße	63,5254 ms
$\frac{1}{64}$ Bildgröße	63,9661 ms
$\frac{1}{256}$ Bildgröße	64,1186 ms
1 Pixel	64 ms

Tabelle 8.3: Laufzeiten für unterschiedlich große Pixelregionen

Durch Unterabtastung kann die Laufzeit auf ca. 63 ms reduziert werden. Denn wie aus den Werten in Tabelle 8.3 zu erkennen ist, stagniert die Laufzeit bei 63 ms, sodass auch eine stärkere Unterabtastung keine Verbesserung mit sich bringt. Gleichzeitig sorgt diese Unterabtastung aber für einen Qualitätsverlust des Ergebnisses. Um einen angemessenen Kompromiss zwischen Laufzeit und Genauigkeit des Algorithmus zu erzielen, wird daher das Bild durch Unterabtastung auf $\frac{1}{16}$ der ursprünglichen Bildgröße reduziert.

Zum anderen besteht auch eine Relation zwischen der Laufzeit dieser Funktion und dem Inhalt des Bildes bzw. der Pixelwerte, die übertragen werden.

In diesem Zusammenhang wurden drei verschiedene Testszenen untersucht und die entsprechende Laufzeit gemessen. Testszene 1 zeigt einen ColorChecker, der von einer Lichtquelle teilweise angestrahlt wird, eine Kugel mit spiegelnder Oberfläche und

Abbildung 8.1: 1. Testszene

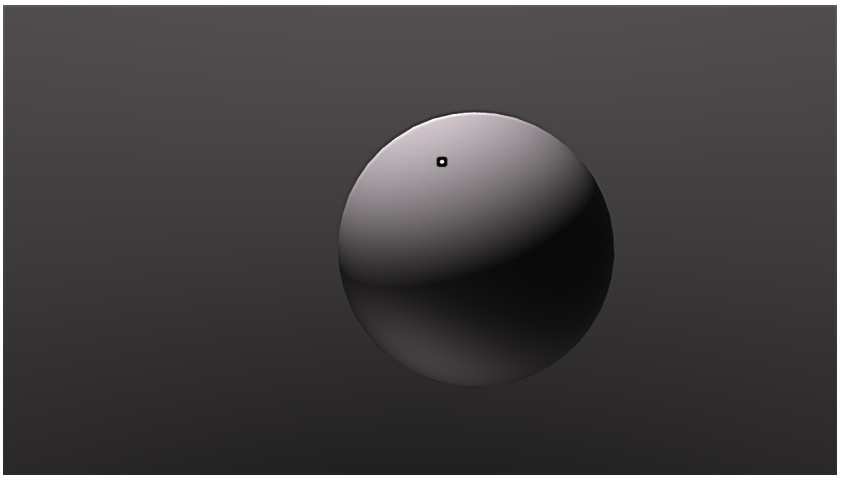


Abbildung 8.2: 2. Testszene

Abbildung 8.3: 3. Testszene

eine Kugel mit diffuser Oberfläche. Als Hintergrund dient ein HDR-Skydome (s. Abbildung 8.1).

Testzene 2 zeigt eine diffuse graue Kugel in grauer Umgebung (s. Abbildung 8.2 auf der vorherigen Seite). Die dritte Testszene zeigt ein rotes Modell eines Affenkopfes mit einem zweiten HDR-Skydome als Hintergrund (s. Abbildung 8.3 auf der vorherigen Seite).

Bei allen drei Testszenen wurden bei vierfacher Unterabtastung pro Dimension die Laufzeiten gemessen. Die Ergebnisse zeigt Tabelle 8.4.

Szene	ø Laufzeit
Szene 1	63,6271 ms
Szene 2	47,4915 ms
Szene 3	64,3220 ms

Tabelle 8.4: Laufzeiten für unterschiedlichen Bildinhalt

8.1.3 Optimierung

Ersetzt man die Berechnung der Luminanzwerte mit dem in Abschnitt 7.1 unter der Überschrift „Optimierung“ vorgestellten Verfahren, kann man den Pixeltransfer vermeiden.

Eine erneute Laufzeitmessung verdeutlicht den Vorteil dieser Optimierung. Die Ergebnisse finden sich in Tabelle 8.5.

Shader	ø Laufzeit
StandardTMO	$1,3898$ ms
ColorCorrectedTMO	$2,31\bar{6}$ ms
MixedTMO	$2,6$ ms
iCam06	$2,3\bar{6}$ ms
CIAR	$2,7\bar{3}$ ms
FilmicTMO	$2,51\bar{6}$ ms
MyTMO	$2,61\bar{6}$ ms
LocalLaplacianTMO	$4,15$ ms

Tabelle 8.5: Laufzeiten der einzelnen Shader, optimiert

Die Laufzeiten fast aller Shader liegen nun unter 3 ms. Dies entspricht einem Geschwindigkeitsverlust von unter 1,5 ms.

Lediglich die Laufzeit des LocalLaplacianTMO reist mit 4,15 ms nach oben aus, was einer Abweichung von ca. 2,8 ms entspricht. Eine Erklärung für diese Abweichung

liefert die Tatsache, dass für den LocalLaplacianTMO deutlich mehr RenderPasses durchgeführt werden müssen als für die anderen Shader (vgl. Kapitel 5 auf Seite 29). Die Anforderungen für Real Time Anwendungen sind jedoch für alle Operatoren erfüllt.

8.2 Qualität

Die Qualität der verschiedenen Verfahren spiegelt sich in ihrer Fähigkeit wieder, den ursprünglichen Farbeindruck trotz Kompression der Luminanz zu erhalten.

Der Farbeindruck ist jedoch subjektiv und damit von Person zu Person unterschiedlich. Um trotzdem eine möglichst eindeutige Aussage darüber treffen zu können, wurde im Rahmen dieser Thesis eine Nutzerstudie durchgeführt. Darin wurden die Ergebnisse dieser Verfahren mehreren Testpersonen präsentiert und von diesen bewertet.

8.2.1 Apparatus

Für die Studie wurde folgende Hardware verwendet:

Komponente	Daten
CPU	Intel i5-4440 Quadcore 3,1Ghz
Arbeitsspeicher	8 GB
Grafikkarte	GeForce GTX 750Ti 128-bit GDDR5 2048MB
Betriebssystem	Windows 8.1 64 Bit
Monitor	Dell Ultrasharp 2412M

Tabelle 8.6: Plattform Nutzerstudie

Die gesamte Studie fand in einem Testraum statt, der grau gestrichene Wände besaß. So sollte die Farbe der Betrachtungsumgebung möglichst neutral gehalten werden. Zudem wurde der Raum über zwei HMI-Lampen konstant mit Tageslicht ausgeleuchtet (vgl. Abbildung 8.4 auf der nächsten Seite).

8.2.2 Nutzerstudie

Aus den implementierten Verfahren wurde zunächst eine Vorauswahl getroffen. Diejenigen Verfahren, die eine deutlich schlechtere Performance oder auf den ersten Blick ersichtliche Qualitätsmängel zeigten, wurden in der Studie nicht berücksichtigt. Es blieben vier Algorithmen: CIAR, der ColorCorrectedTMO, der FilmicTMO und der selbst entwickelte Operator. Diese Verfahren werden durch einen Paarvergleich bewertet. Die vier Verfahren wurden in insgesamt sechs Anordnungen gegenübergestellt, sodass jedes mit jedem anderen verglichen wurde.

Abbildung 8.4: Testraum

An der Studie nahmen zwölf Testpersonen teil. Diese haben die Evaluierung jeweils einzeln durchgeführt, damit die Ergebnisse unbeeinflusst blieben. Die Vorgehensweise lief dann immer gleich ab:

Eine Testperson nimmt vor dem oben genannten Monitor Platz. Auf dem Monitor ist das Live-Rendering zweier Szenen mit je zwei ColorCheckern zu sehen, welches von der Testperson selbst gesteuert werden kann. Das gleiche Fenster zeigt zudem die Textur des ColorCheckers am rechten Bildrand als Referenz. Abbildung 8.5 zeigt das komplette Programmfenster zur Veranschaulichung.

Abbildung 8.5: Programmfenster für die Nutzerstudie

Über die Buttons am rechten Bildrand kann die Testperson dann den Paarvergleich wechseln. Dabei ist pro Vergleich immer nur ein Verfahren gleichzeitig aktiv, welches über die Buttons *Verfahren 1* und *Verfahren 2* gewechselt werden kann. Dadurch wird gewährleistet, dass die Verfahren immer aus der gleichen Perspektive betrachtet werden.

Eine Abweichung im Betrachtungswinkel könnte sonst in Abhängigkeit vom Monitor für Farbabweichungen sorgen, die das Ergebnis verfälschen könnten.

Mit Bedienung der Buttons *Szene 1* und *Szene 2* hat die Testperson zudem die Möglichkeit, die jeweilige Szene zu wechseln.

Jeder Testperson wird ein Fragebogen vorgelegt, um die Ergebnisse zu erfassen. Dieser wird während der Laufzeit ausgefüllt.

Fragen / Bewertungskriterien

Der verwendete Fragebogen ist auf der CD im Anhang zu dieser Thesis zu finden. Er beinhaltet zu jedem Paarvergleich fünf identische Fragen.

Die erste Frage bezieht sich auf die Sichtbarkeit von Details in hellen und dunklen Bereichen. Sie dient als Indiz für die Fähigkeit des Algorithmus, Helligkeiten zu komprimieren. Sind keine Details sichtbar, ist die Kompression nicht stark genug.

Frage 2 untersucht die Sichtbarkeit statischer Artefakte. Diese können entweder durch das Verfahren selbst oder eine unzureichende Implementierung entstehen und stören in der Regel den Gesamteindruck des Bildes.

Ähnlich dazu ist Frage 3. Hier geht es jedoch um Bewegungsartefakte, die nur bei der Anwendung des Algorithmus auf Bildsequenzen entstehen. Bei einzelnen Bildern sind diese Effekte nicht vorhanden. Sie haben jedoch einen ähnlichen negativen Einfluss auf die empfundene Qualität wie statische Artefakte.

In der vierten Frage soll dann explizit die Referenztextur herangezogen werden. Anhand dieser kann angegeben werden, wie gut ein Verfahren deren Farbe nach dem Rendering reproduziert. Mit dieser Frage kann die Farberscheinung qualitativ bewertet werden.

Als fünftes wird schließlich nach dem Gesamteindruck des Renderings auf den Betrachter gefragt. Im Zusammenspiel mit der ersten und vierten Frage kann so darauf geschlossen werden, ob für den Gesamteindruck dem Betrachter eher die Farbwiedergabe oder die Luminanzkompression ausschlaggebend ist.

Grundsätzlich soll als Antwort das Verfahren angegeben werden, welches von den beiden Möglichkeiten das bessere Ergebnis liefert.

Auswertung

Zur Auswertung der Nutzerstudie wurde ein zusammengesetztes Punktesystem eingesetzt: Bei der ersten, vierten und fünften Frage wurde ein 2-Punktesystem zur Auswertung herangezogen. Für jedes Kriterium, bei dem ein Verfahren besser als der Vergleich abschnitt, wurde diesem Verfahren zwei Punkte zugeteilt. Das schlechtere Verfahren erhielt null Punkte. Bei einem Unentschieden erhielten beide Verfahren jeweils einen Punkt.

Dieses System wurde für die zweite und dritte Frage als nicht ausreichend erachtet, da so keine quantitative Aussage über die sichtbaren Artefakte getroffen werden konnte. Bei diesen Fragen kam eine zusätzliche Bewertungsskala zum Einsatz, anhand derer die Testpersonen die Auffälligkeit der Artefakte einschätzen sollten. Traten Artefakte in großer Anzahl oder an sehr auffälligen Stellen auf, erhielt das Verfahren eine niedrige Punktzahl. Je weniger Artefakte sichtbar waren, desto mehr Punkte wurden verteilt. Die

Testpersonen konnten so bis zu vier Punkte pro Verfahren pro Frage vergeben.

Mit einem solchen System lässt sich für jedes Verfahren pro Kriterium eine Durchschnittspunktzahl berechnen. Je größer diese ausfällt, desto besser hat das Verfahren abgeschnitten. Durch die Verwendung von verschiedenen Maximalpunktzahlen sind die einzelnen Kriterien untereinander schwer zu vergleichen. Daher wurde für jedes Kriterium die erreichte Prozentzahl der jeweils maximal möglichen Punktzahl berechnet. So können die Kriterien verglichen und auch Aspekte identifiziert werden, bei denen die Meinungen der Testpersonen stark auseinander gehen.

8.2.3 Sichtbarkeit von Details

Bezüglich der Sichtbarkeit von Details ergibt sich bei allen Verfahren ein ähnliches Bild. Tabelle 8.7 zeigt die durchschnittliche Punktzahl.

Shader	ø Punkte (von 2)	Prozent
ColorCorrectedTMO	$0,97\overline{2}$	$48,6\overline{1}$
CIAR	1	50
FilmicTMO	$0,91\overline{6}$	$45,8\overline{3}$
MyTMO	$1,\overline{1}$	$55,\overline{5}$

Tabelle 8.7: Durchschnittspunktzahlen für die Darstellung von Details

Es ist erkennbar, dass MyTMO das beste Ergebnis hat. Danach folgen CIAR und der ColorCorrectedTMO. Der filmische Operator bildet das Schlusslicht. Alle Verfahren liegen aber sehr nah beieinander. So beträgt der Abstand zwischen dem letzten und dem besten Platz lediglich 9,7%. Die Fähigkeit, die Helligkeit des Bildes zu komprimieren, scheint somit bei allen Verfahren annähernd gleich gut zu sein.

8.2.4 Statische Artefakte

Bei den statischen Artefakten sieht das Ergebnis anders aus. Hier liegt der filmische Operator deutlich vorne. Der Abstand zum Zweitplatzierten beträgt fast 25%. Tabelle 8.8 auf der nächsten Seite zeigt die Ergebnisse zu diesem Kriterium.
Die übrigen drei Verfahren liegen dagegen näher beieinander. Hier beträgt der maximale Unterschied nur ca. 6%.

Eine mögliche Erklärung hierfür liegt in der Tatsache, dass der filmische Operator der einzige rein globale Operator ist. Lokale Operatoren dagegen sind anfälliger für Artefakte, welche aus der Berechnung der Nachbarschaftsbeziehungen zwischen den Pixeln entstehen (siehe Abschnitt 5.2 auf Seite 31).

Zu beachten ist, dass bei dieser Frage einer der Fragebögen lückenhaft ausgefüllt wurde. Er wurde daher nicht in die Bewertung dieses Kriteriums mit einbezogen. Der Punktedurchschnitt wurde hier also nur über 11 Testpersonen ermittelt.

Shader	ø Punkte (von 4)	Prozent
ColorCorrectedTMO	$2,58\bar{3}$	$64,58\bar{3}$
CIAR	$2,8\bar{3}$	$70,8\bar{3}$
FilmicTMO	$3,\bar{8}1$	$95,\bar{4}5$
MyTMO	$2,\bar{6}3$	$65,\bar{9}0$

Tabelle 8.8: Durchschnittspunktzahlen für die Anfälligkeit für statische Artefakte

8.2.5 Bewegungsartefakte

In Bezug auf die Sichtbarkeit von Bewegungsartefakten sind sich die Verfahren wieder ähnlicher. Auch hier liegt der FilmicTMO mit 3,61 Punkten vorne. MyTMO wurde als Schlusslicht allerdings immer noch mit durchschnittlich 3,27 Punkten bewertet und liegt damit ca. 8 Prozentpunkte hinter dem filmischen Operator. Wie in Tabelle 8.9 zu sehen ist, liegt die Punktzahl bei allen Verfahren deutlich über drei Punkten bzw. über 80% und ist damit relativ hoch. Somit können alle Verfahren als relativ robust gegen Bewegungsartefakte angesehen werden.

Shader	ø Punkte (von 4)	Prozent
ColorCorrectedTMO	$3,3\bar{8}$	$84,7\bar{2}$
CIAR	$3,41\bar{6}$	$85,41\bar{6}$
FilmicTMO	$3,6\bar{1}$	$90,2\bar{7}$
MyTMO	$3,\bar{2}7$	$81,9\bar{4}$

Tabelle 8.9: Durchschnittspunktzahlen für die Anfälligket für Bewegungsartefakte

8.2.6 Farbwiedergabe

Die Ergebnisse der Bewertung der Farbwiedergabe in Tabelle 8.10 zeigen ein eindeutiges Ergebnis.

Shader	ø Punkte (von 2)	Prozent
ColorCorrectedTMO	$1,25$	$62,5$
CIAR	$0,9\bar{4}$	$47,\bar{2}$
FilmicTMO	$1,\bar{4}5$	$72,\bar{7}2$
MyTMO	$0,\bar{4}5$	$22,\bar{7}2$

Tabelle 8.10: Durchschnittspunktzahlen für die Darstellung von Farben

Hier liegt erneut der filmische Operator vorne, gefolgt vom ColorCorrectedTMO. Es folgen CIAR und mit deutlichem Abstand MyTMO. Letzterer ist mit durchschnittlich 0,45 Punkten klar abgeschlagen.

Auch diese Frage wurde auf einem der Fragebögen nicht beantwortet, sodass sich die Ergebnisse auf 11 Testpersonen beziehen.

8.2.7 Natürlichkeit des Ergebnisses

Ein ebenso klares Ergebnis liegt bei der Bewertung der Natürlichkeit des Bildes vor. Die zugehörigen Punktzahlen zeigt Tabelle 8.11.

Shader	ø Punkte (von 2)	Prozent
ColorCorrectedTMO	$1,02\overline{7}$	$51,3\overline{8}$
CIAR	$0,\overline{6}$	$33,\overline{3}$
FilmicTMO	$1,41\overline{6}$	$70,8\overline{3}$
MyTMO	$0,\overline{8}$	$44,\overline{4}$

Tabelle 8.11: Durchschnittspunktzahlen für die Natürlichkeit des Ergebnisses

Auch hier liegt der filmische Operator deutlich vorne. Es folgt der ColorCorrectedTMO, der durchschnittlich etwa die Hälfte der möglichen Punkte erzielte. Den letzten Platz belegt dieses Mal CIAR mit $0,\overline{6}$ Punkten hinter MyTMO mit $0,\overline{8}$ Punkten.

Gerade bei dieser Frage ist ein solches eindeutiges Ergebnis überraschend. Die Bewertung hängt hier stark von der persönlichen Erfahrung und Vorstellung der jeweiligen Testperson ab. Dies führt normalerweise zu stark unterschiedlichen Bewertungen und damit zu einem insgesamt ausgeglichenerem Ergebnis mit geringeren Abständen zwischen den Verfahren. Da dies nicht der Fall ist, ist davon auszugehen, dass die Qualitätsunterschiede so stark sind, dass im Vergleich hierzu die Unterschiede im Geschmack der einzelnen Testpersonen in den Hintergrund treten.

8.2.8 Gesamtwertung

Alles in Allem ergeben sich insgesamt die Punktzahlen in Tabelle 8.12 auf der nächsten Seite.

Der FilmicTMO liegt mit ca. 11,2 Punkten vorne. Es folgen der ColorCorrectedTMO und CIAR. Dahinter liegt MyTMO mit 8,37 Punkten auf dem letzten Platz.

Betrachtet man die dazugehörigen Prozentzahlen der Maximalpunktzahl von 14 Punkten, ist erkennbar, dass die letzten drei Verfahren nah beieinander liegen. Der Abstand zum ersten Platz beträgt jedoch über 14 Prozent.

Shader	ø Punkte (von 14)	Prozent
ColorCorrected TMO	$9,\bar{2}$	$65,87$
CIAR	$8,86\bar{1}$	$63,29$
Filmic TMO	$11,21\bar{7}1$	$80,12$
MyTMO	$8,37$	$59,77$

Tabelle 8.12: Durchschnittspunktzahlen für die Gesamtqualität

9 Fazit

9.1 Zusammenfassung der Ergebnisse

In den vorherigen Abschnitten wurde auf die Problematik bei der Darstellung von HDR-Bildern und die zu diesem Zweck entwickelten Tone Mapping Algorithmen eingegangen. Es wurden sieben Verfahren vorgestellt und untersucht, die das Tone Mapping um eine Korrektur der Farberscheinung erweitern, sodass Farben mit konstanter Erscheinung wiedergegeben werden können.

Die Auswertung dieser Verfahren lässt sich wie folgt zusammenfassen:

Die Performance der Algorithmen ist insgesamt gut. Jedes der Verfahren eignet sich in der implementierten Version für eine Nutzung in Echtzeitanwendungen. Dabei ist auch der Laufzeitunterschied zwischen den Verfahren mit unter 1,5 ms relativ gering. Es kann davon ausgegangen werden, dass andere Aspekte des Rendering-Prozesses im Hinblick auf Latenzen deutlich kritischer sind.

Für dieses Ergebnis ist es jedoch unbedingt erforderlich, dass sämtliche Teile des jeweiligen TMOs durch Shader implementiert werden. Wird zwischendurch auf die CPU zurückgegriffen, steigen die Laufzeiten extrem an. In diesem Fall eignet sich nur der filmische Operator für eine Anwendung mit Echtzeitanforderung.

Dieser Operator erzielt auch im Hinblick auf die Qualität die besten Ergebnisse. Er wurde bei allen gegebenen Kriterien mit Ausnahme der Sichtbarkeit von Details am besten bewertet. Die Resultate sind hier eindeutiger bzw. die Unterschiede zwischen den Algorithmen deutlich größer als bei der Laufzeituntersuchung. Das zeigt sich vor allem bei der Bewertung der Farberscheinung, die für diese Thesis eine besondere Relevanz hat.

Auffällig ist, dass der FilmicTMO der einzige rein globale Operator in der Gegenüberstellung ist. Bedenkt man die Vor- und Nachteile von globalen und lokalen Operatoren (vgl. Kapitel 5 auf Seite 29), lässt das den Schluss zu, dass die Existenz von Artefakten den Gesamteindruck stärker beeinflusst, als die Fähigkeit eines Operators zur Luminanzkompression. Dies müsste jedoch in weiteren Untersuchungen geklärt werden.

Denkbar wäre auch eine Kombination aus der Tone Mapping Funktion des MyTMOs und der Farbverarbeitung des FilmicTMOs, um die beide Vorteile in einem neuen Operator zu vereinen.

Insgesamt ist an dieser Stelle der FilmicTMO für die weitere Verwendung zu empfehlen.

9.2 Ausblick

Die Auswertung wirft jedoch auch einige Fragen für zukünftige Untersuchungen auf.

Das ist zum einen die bereits angesprochene Frage, welches der in der Studie verwendeten Kriterien die wichtigere Rolle bei der Bewertung des Gesamteindrucks spielt. In diesem Zusammenhang ist ein Aspekt interessant, der während der Studie von einem der Probanden angesprochen wurde: Bei einer bestimmten Kameraperspektive trat am Rand des ColorCheckers durch Unterabtastung Aliasing auf, das durch das Rendering hervorgerufen wurde. Diese Störung war aber nur bei denjenigen Algorithmen sichtbar, die sich durch eine gute Darstellung von Details auszeichneten. Bei den anderen Verfahren wurde es durch das Tone Mapping unterdrückt. In diesen Fällen führte also eine schlechtere Luminanzkompression zu einem besseren Endergebnis.

Dies zeigt, wie komplex sich die Wahl des passenden Algorithmus für eine bestimmte Anwendung gestaltet. So ergibt sich zum anderen auch die Frage, ob es überhaupt einen einzigen Tone Mapping Algorithmus geben kann, der für alle Anwendungen die optimalen Ergebnisse liefert.

Für die nahe Zukunft stellt sich im Hinblick auf die ermittelten Laufzeiten die Frage, wie sich die Algorithmen bei einer deutlich erhöhten Auflösung verhalten. Denn die Testergebnisse wurden in dieser Thesis für Bilder mit einer Auflösung von 1280 x 720 Pixeln ermittelt. Diese Auflösung ist relativ klein, betrachtet man die Entwicklung hin zu 4K-Auflösung im Kino und Internet bzw. sogar 8K im Fernsehbereich im Vergleich.

Höhere Auflösungen könnten sich aber nicht nur auf die Laufzeit auswirken. Es liegt nahe, dass der Zuschauer bei einer höheren Auflösung auch mehr Details und damit eine bessere Luminanzkompression erwartet. Auch die Akzeptanz von Artefakten könnte in diesem Fall sinken, da diese mehr ins Gewicht fielen.

Schließlich bildet auch die Implementierung einer vollständig lokalen Version des CIAR-Verfahrens einen interessanten Aspekt für weitergehende Untersuchungen. Die Einteilung des Bildes in Regionen mit annähernd gleicher Beleuchtung erscheint als vielversprechender Ansatz sowohl für eine genauere Luminanzkompression als auch eine präzisere Farberscheinung.

10 Danksagung

Diese Thesis wäre ohne die Unterstützung einiger Personen nicht möglich gewesen. Daher möchte ich diesen Personen hier meinen herzlichen Dank aussprechen.

Ein besonderer Dank möchte ich an Herrn Prof. Fuhrmann richten für die Betreuung meiner Thesis, den vielen guten Ratschlägen und die konstruktive Kritik.
Ebenso sehr bedanken möchte ich mich auch bei Herrn Prof. Fischer, der sich als Zweitprüfer für meine Thesis bereit gestellt und mich ebenfalls mit einigen entscheidenden Ratschlägen unterstützt hat.

Daneben möchte ich mich auch bei Tobias Bayer und der ganzen Computer Graphics Group der FH Köln bedanken für die vielen Hilfestellungen zum Renderer.

Ein Dank geht auch an Nadine Wiehlpütz, Sylvia Moritz und Michael Schuff für die großzügige Bereitstellung des Fotoateliers und die Unterstützung während der Nutzerstudie.

Diese Studie wäre natürlich nicht umsetzbar gewesen ohne die Probanden, die bereitwillig ihre Zeit geopfert haben, um mich zu unterstützen. Auch dafür möchte ich mich herzlich bedanken.

Zu guter Letzt möchte ich auch einen großen Dank an meine Eltern richten. Sie haben mir durch viele gute Ratschläge und Ermunterungen oft den Rücken gestärkt.

Literatur

[1] Ansel Adams und Robert Baker. *The print*. 1983, 210 p.

[2] Ahmet Oguz Akyüz und Erik Reinhard. "Color appearance in high-dynamic-range imaging". In: *Journal of Electronic Imaging* 15.3 (Juli 2006), S. 033001.

[3] *AMP HDR Kamera*. URL: http://amphdr.com/ (besucht am 02. 03. 2015).

[4] Torge Anders. *CIE-Normfarbtafel*. URL: http://commons.wikimedia.org/wiki/File:CIE-Normfarbtafel.png (besucht am 03. 05. 2015).

[5] Mathieu Aubry u. a. "Fast and robust pyramid-based image processing". In: *CSAIL Tech Report* (2011).

[6] Mathieu Aubry u. a. "Fast Local Laplacian Filters : Theory and Applications". In: *ACM Trans. on Graphics* VV (2014), S. 15.

[7] Blackmagic. *Blackmagic Pocket Cinema Camera*. URL: https://www.blackmagicdesign.com/products/blackmagicpocketcinemacamera (besucht am 20. 02. 2015).

[8] F J Blommaert und J B Martens. "An object-oriented model for brightness perception." In: *Spatial vision* 5.1 (1990), S. 15–41.

[9] R R Buckley. "Reproducing pictures with non-reproducible colors". Diss. MIT, 1978.

[10] Dominic Case. *Film Technology in Post Production*. Oxford: Focal Press, 2012, S. 224.

[11] CIE. *Guidelines for the Evaluation of Gamut Mapping Algorithms*. CIE. Commission internationale de l'Eclairage, CIE Central Bureau, 2004.

[12] P. E. Debevec und J. Malik. "Recovering high dynamic range radiance maps from photographs. SIGGRAPH 97 Conf". In: *Proc., August* (1997), S. 3–8.

[13] Paul Debevec. "Image-Based Lighting". In: *IEEE Computer Graphics and Applications* April (2002).

[14] Paul Debevec. *Light Probe Image Gallery*. URL: http://pauldebevec.com/Probes/ (besucht am 14. 05. 2015).

[15] Paul Debevec. "Rendering Synthetic Objects into Real Scenes : Bridging Traditional and Image-based Graphics with Global Illumination and High Dynamic Range Photography". In: *In Computer Graphics Proceedings, Annual Conference Series (Proc. ACM SIGGRAPH '98 Proceeding)* (1998), S. 189–198.

[16] Robert L. Donofrio. “Review Paper: The Helmholtz-Kohlrausch effect”. In: *Journal of the Society for Information Display* 19.10 (2011), S. 658.

[17] Frédo Durand und Julie Dorsey. “Fast bilateral filtering for the display of high-dynamic-range images”. In: *ACM Transactions on Graphics* 21 (2002).

[18] David S Ebert u. a. *Texturing and modeling: a procedural approach*. 3rd Editio. Morgan Kaufmann Publishers Inc., 2002, S. 1–721.

[19] Fritz Ebner und Mark D Fairchild. “Development and Testing of a Color Space (IPT) with Improved Hue Uniformity”. In: *Color Imaging Conference*. 1998, S. 8–13.

[20] M D Fairchild. *Color Appearance Models*. Chichester, UK: John Wiley & Sons, Ltd, Aug. 2005, S. 1–409.

[21] Mark D. Fairchild und Garrett M. Johnson. “iCAM framework for image appearance, differences, and quality”. In: *Journal of Electronic Imaging* 13.1 (Jan. 2004), S. 126.

[22] Raanan Fattal, Dani Lischinski und Michael Werman. “Gradient domain high dynamic range compression”. In: *ACM Transactions on Graphics* 21 (2002).

[23] Jan Froehlich. *Creating cinematic wide gamut HDR-video for the evaluation of tone mapping operators and HDR-displays*. 2014.

[24] Jan Fröhlich u. a. *HDM-HDR-2014 HDR-Video data set*. 2014. URL: `https://hdr-2014.hdm-stuttgart.de/` (besucht am 11. 05. 2015).

[25] GoHDR. *goHDR Video C2*. URL: `http://gohdr.com/website/gohdr-products/` (besucht am 02. 03. 2015).

[26] John Hable. *Filmic Tonemapping Operators*. 2010. URL: `http://filmicgames.com/archives/75` (besucht am 14. 04. 2015).

[27] Kai Hamann. *So funktionieren Kamera-Automatikprogramme*. 2007. URL: `http://www.pc-magazin.de/ratgeber/so-funktionieren-kamera-automatikprogramme-86547.html` (besucht am 11. 02. 2015).

[28] Jürgen Held. *HDR-Fotografie. Das umfassende Handbuch*. 2. erweite. Galileo Design, 2009, S. 368.

[29] R W G Hunt. “Light and dark adaptation and the perception of color.” In: *Journal of the Optical Society of America* 42.3 (1952), S. 190–199.

[30] R. W. G. Hunt. *The Reproduction of Colour*. Bd. 37. The Wiley-IS&T Series in Imaging Science and Technology. Wiley, 1969, S. 114.

[31] ISO. *ISO 3664:2009, Graphic technology and photography – Viewing conditions*. 2009.

[32] Sing Bing Kang u. a. “High dynamic range video”. In: *ACM Transactions on Graphics* 22 (2003), S. 319.

[33] Naoya Katoh und Masahiko Ito. *Gamut Mapping for Computer Generated Images*. 1995.

[34] Bruno Keller u. a. *Leuchtdichte*. URL: http://www.bph.hbt.arch.ethz.ch/Filep/Licht/Licht%5C_Grundl/Kennwerte/Leuchtdichte.html (besucht am 11. 02. 2015).

[35] Douglas a Kerr. “The CIE XYZ and xyY Color Spaces”. In: 1 (2010), S. 1–16.

[36] Richard Kirk. “Standard RGB color spaces”. 2010.

[37] Jiangtao Kuang, Garrett M. Johnson und Mark D. Fairchild. “iCAM06: A refined image appearance model for HDR image rendering”. In: *Journal of Visual Communication and Image Representation* 18.5 (Okt. 2007), S. 406–414.

[38] E H Land. “An alternative technique for the computation of the designator in the retinex theory of color vision.” In: *Proceedings of the National Academy of Sciences of the United States of America* 83.10 (1986), S. 3078–3080.

[39] E H Land. “Recent advances in retinex theory and some implications for cortical computations: color vision and the natural image.” In: *Proceedings of the National Academy of Sciences of the United States of America* 80.16 (1983), S. 5163–5169.

[40] Magic Lantern. *Magic Lantern User’s Guide*. URL: http://wiki.magiclantern.fm/userguide%5C#hdr-video (besucht am 20. 02. 2015).

[41] Gregory Ward Larson, Holly Rushmeier und Christine Piatko. “A visibility matching tone reproduction operator for high dynamic range scenes”. In: *IEEE Transactions on Visualization and Computer Graphics* 3 (1997), S. 291–306.

[42] Leica. *Leica S-System*. URL: http://de.leica-camera.com/Photography/Leica-S/About-the-S-System (besucht am 20. 02. 2015).

[43] M. R. Luo und C. Li. “CIECAM02 and Its Recent Developments”. In: *Advanced Color Image Processing and Analysis*. 2013, S. 19–58.

[44] LWJGL. *Light Weight Java Gaming Library 3*. URL: http://www.lwjgl.org/ (besucht am 11. 05. 2015).

[45] Lindsay MacDonald, Kaida Xiao und Ján Morovic. “A Topographic Gamut Compression Algorithm”. In: *The Journal of Imaging Science and Technology* 46.3 (2002), S. 228–236.

[46] S Mann und R W Picard. “On Being ‘undigital’ With Digital Cameras: Extending Dynamic Range By Combining Differently Exposed Pictures”. In: *Proceedings of IS&T* (1995), S. 442–448.

[47] D Marr und E Hildreth. “Theory of edge detection.” In: *Proceedings of the Royal Society of London. Series B, Containing papers of a Biological character. Royal Society (Great Britain)* 207.1167 (1980), S. 187–217.

[48] John J McCann. “Color gamut measurements and mapping: The role of color spaces”. In: *Color Imaging: Device-Independent Color, Color Hardcopy, and Graphic Arts IV* SPIE 3648 (1999), S. 68–82.

[49] J Meyer und B Barth. “Color Gamut Matching for Hard Copy”. In: *Proc. of SID Digest* (1989), S. 86–89.

[50] T. Mitsunaga und S.K. Nayar. “Radiometric self calibration”. In: *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)* 1 (1999), S. 374–380.

[51] Ján Morovič. *Color Gamut Mapping*. Wiley, 2009, S. 1–287.

[52] L Neumann und A Neumann. “Gamut clipping and mapping based on the coloroid system”. In: *Proc. CGIV 2nd Eur. Conf. Color in Graphics, Imaging, and Vision* (2004), S. 548–555.

[53] Alan R. Robertson Noburo Ohta. *Colorimetry - Fundamentals and Applications*. 2006.

[54] Okun. *The VES Handbook of Visual Effects*. 2010, S. 335–386.

[55] Otoy. *Octane Renderer*. URL: http://home.otoy.com/render/octane-render/ (besucht am 14. 05. 2015).

[56] S Paris, Sw Hasinoff und J Kautz. “Local Laplacian filters: edge-aware image processing with a Laplacian pyramid”. In: *ACM Trans. Graph.* (2011).

[57] Sylvain Paris, Samuel W Hasinoff und Jan Kautz. *Local Laplacian Filters: Edge-aware Image Processing with a Laplacian Pyramid (Matlab-Code)*. URL: http://people.csail.mit.edu/sparis/publi/2011/siggraph/ (besucht am 09. 04. 2015).

[58] Tania Pouli. *Calibtrated Image Appearance reproduction (Matlab Code)*. URL: http://taniapouli.co.uk/research/image%5C_appearance/ (besucht am 09. 04. 2015).

[59] Tania Pouli, Alessandro Artusi und Francesco Banterle. “Color Correction for Tone Reproduction”. In: *Color and Imaging Conference* (2013), S. 215–220.

[60] William K. Pratt. *Digital Image Processing: PIKS Scientific Inside*. Bd. 5. Wiley, 2006, S. 738.

[61] D. McL. Purdy. “Spectral Hue as a Function of Intensity OF”. In: *The American Journal of Psychology* 43 (1931), S. 541–559.

[62] Z.-U. Rahman, D J Jobson und G W Woodell. “Multiscale retinex for color rendition and dynamic range compression”. In: *Applications of Digital Image Processing XIX* SPIE-2847 (1996), S. 183–191.

[63] Erik Reinhard. *Color Imaging: Fundamentals and Applications*. Ak Peters Series. Taylor & Francis, 2008.

[64] Erik Reinhard und Kate Devlin. “Dynamic range reduction inspired by photoreceptor physiology”. In: *IEEE Transactions on Visualization and Computer Graphics* 11.1 (2005), S. 13–24.

[65] Erik Reinhard u. a. “Calibrated image appearance reproduction”. In: *ACM Transactions on Graphics* 31.6 (2012), S. 1.

[66] Erik Reinhard u. a. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting (The Morgan Kaufmann Series in Computer Graphics)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[67] Erik Reinhard u. a. “Photographic tone reproduction for digital images”. In: *ACM Transactions on Graphics* 21 (2002).

[68] Geoff Richards. *BrightSide DR37-P Monitor*. URL: http://www.bit-tech.net/hardware/2005/10/04/brightside%5C_hdr%5C_edr/1 (besucht am 11. 05. 2015).

[69] Mark a. Robertson, Sean Borman und Robert L. Stevenson. “Estimation-theoretic approach to dynamic range enhancement using multiple exposures”. In: *Journal of Electronic Imaging* 12.April (2003), S. 219.

[70] Jacob Rus. *HSL-Farbraum*. 2010. URL: http://en.wikipedia.org/wiki/File:HSL%5C_color%5C_solid%5C_dblcone%5C_chroma%5C_gray.png (besucht am 15. 12. 2014).

[71] Jason J Sara. “The automated reproduction of pictures with nonreproducible colors”. Diss. MIT, 1984, S. 177.

[72] Daniel Schilberg, Tobias Meisen und Rudolf Reinhard. “Virtuelle Produktion – Die Virtual Production Intelligence im Einsatz”. In: *Exploring Virtuality* (2014), pages.

[73] SpheronVR. *HDRv introduction*. URL: https://www.spheron.com/fileadmin/media/05%5C_About%5C_Us/56%5C_History/HDRv/SpheronVR%5C_%5C_HDRv%5C_%5C_introduction%5C_005.pdf (besucht am 01. 03. 2015).

[74] SpheronVR. *SpheroCam HDR*. 2012. URL: http://www.spheron.com/en/spheron-cgi/products/spherocam-hdr.html (besucht am 20. 02. 2015).

[75] *sRGB-Farbraum*. URL: http://en.wikipedia.org/wiki/File:CIExy1931%5C_srgb%5C_gamut.png (besucht am 15. 12. 2014).

[76] J C Stevens und S S Stevens. “Brightness function: effects of adaptation.” In: *Journal of the Optical Society of America* 53.3 (1963), S. 375–385.

[77] Jr. Stockham, T.G. “Image processing in the context of a visual model”. In: *Proceedings of the IEEE* 60.7 (1972).

[78] Michael Stokes u. a. *A standard default color space for the internet-srgb*. Techn. Ber. 1996, S. 1–17.

[79] Horand Störmer. *Binary functions and their applications*. Bd. 348. Springer Science & Business Media, 2012.

[80] The Khronos Group. *OpenGL Dokumentation*. URL: https://www.opengl.org/documentation/ (besucht am 19. 04. 2015).

[81] Thomas Akenine-Möller, Eric Haines und Naty Hoffman. *Real-Time Rendering*. 3rd. Boca Raton: CRC Press, 2008.

[82] C. Tomasi und R. Manduchi. “Bilateral filtering for gray and color images”. In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)* (1998).

[83] Jack Tumblin und Greg Turk. "{LCIS}: A boundary hierarchy for detail-preserving contrast reduction". In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), S. 83–90.

[84] B Wissinger, S Kohl und U Langenbeck. *Genetics in Ophthalmology*. Developments in ophthalmology. Karger, 2003.

Eidesstattliche Erklärung

Ich versichere hiermit, die vorgelegte Arbeit in dem gemeldeten Zeitraum ohne fremde Hilfe verfasst und mich keiner anderen als der angegebenen Hilfsmittel und Quellen bedient zu haben.

Köln, den 27. Mai 2015

Unterschrift:

(Vorname, Nachname)